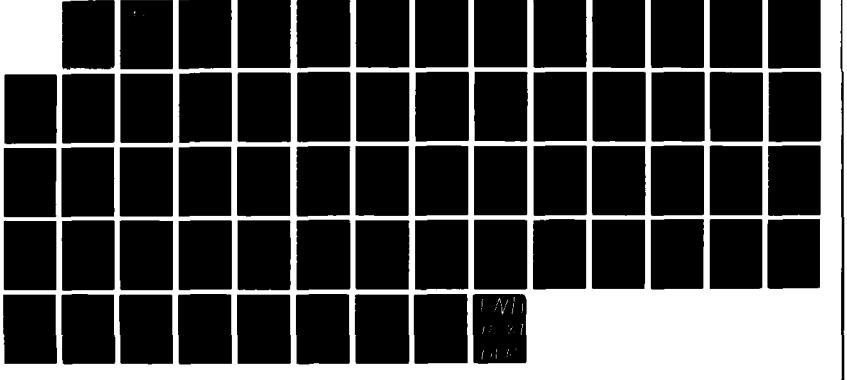
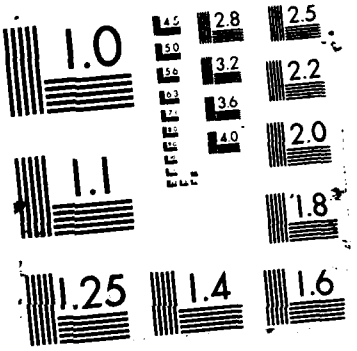
AD-A185 595 FLEXIBLE PARSING(U) CARNEGIE-MELLON UNIV PITTSBURGH PA 1/1
DEPT OF COMPUTER SCIENCE P J HAYES 30 JUN 86
AFOSR-TR-87-1187 $AFOSR-82-0219

UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

UNCLASSIFIED

(2)

AD-A185 595

REPORT DOCUMENTATION PAGE

DTIC FILE COPY

DTIC ELECTE OCT 0 1 1987 S D

1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 87-1187
6a. NAME OF PERFORMING ORGANIZATION: Carnegie Mellon University	6b. OFFICE SYMBOL (If applicable):
6c. ADDRESS (City, State and ZIP Code): Pittsburgh, Pennsylvania 15213-3890	7a. NAME OF MONITORING ORGANIZATION: AFOSR/NM
	7b. ADDRESS (City, State and ZIP Code): AFOSR/NM, Bldg 410, Bolling AFB DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR	8b. OFFICE SYMBOL (If applicable): NM
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR 82-0219
8c. ADDRESS (City, State and ZIP Code): AFOSR/NM, Bldg 410, Bolling AFB DC 20332-6448	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A3	

11. TITLE (Include Security Classification): Flexible Parsing

12. PERSONAL AUTHOR(S): Philip L. Hayes

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 2/1/82 TO 6/30/86		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The research objectives for this effort included the development of flexible parsing techniques, the investigation of design choices for flexible parsers, and the development of flexible parsing techniques for interfaces to interactive computer systems. Thirteen research publications resulted from this grant, including such titles as "Representation of task-independent knowledge in a gracefully interacting user interface," "Dynamic strategy selection in flexible parsing," and "Parsing spoken language: a semantic case frame approach."

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Capt. John Thomas	(202)767-5026	NM

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

87 9 24 027

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

INSTRUCTIONS FOR PREPARATION OF REPORT DOCUMENTATION PAGE

GENERAL INFORMATION

The accuracy and completeness of all information provided in the DD Form 1473, especially classification and distribution limitation markings, are the responsibility of the authoring or monitoring DoD activity.

Because the data input on this form will be what others will retrieve from DTIC's bibliographic data base or may determine how the document can be accessed by future users, care should be taken to have the form completed by knowledgeable personnel. For better communication and to facilitate more complete and accurate input from the origination of the form to those processing the data, space has been provided for the name, telephone number and office symbol of the DoD person responsible for the input cited on the form. These are to be noted in Block 22.

All information on the DD Form 1473 should be typed.

Only information appearing on or in the report, or applying specifically to the report in hand should be reported. If there is any doubt, the block should be left blank.

Some of the information on the forms *(e.g., title, abstract)* will be machine-indexed. The terminology used should describe th content of the report or identify it as precisely as possible for future identification and retrieval.

SPECIAL NOTE: UNCLASSIFIED ABSTRACTS AND TITLES DESCRIBING CLASSIFIED DOCUMENTS MAY APPEAR SEPARATELY FROM THE DOCUMENTS IN AN UNCLASSIFIED CONTEXT, E.G., IN DTIC ANNOUNCEMENT BULLETINS AND BIBLIOGRAPHIES OR OR BY ACCESS IN AN UNCLASSIFIED MODE TO THE RDT/E ON-LINE SYSTEM. THIS MUST BE CONSIDERED IN THE PREPARATION AND MARKING OF UNCLASSIFIED ABSTRACTS AND TITLES.

The Defense Technical Information Center *(DTIC)* is ready to offer assistance to anyone who needs and requests it. Call Data Base Input Division *(AUTOVON)* 284-7044; Com 202-274-7044.

SPECIFIC BLOCKS

SECURITY CLASSIFICATION OF THE FORM:

In accordance with DoD 5200.1-R, Information Security Program Regulation, Chapter V1 Section 2, paragraph 4-200, classification markings are to be stamped, printed, or written at the tops and bottom of the form in capital letters that are larger than those used in the text of the document. See also DoD 5220.22-M, Industrial Security Manual for Safeguarding Classified Information, Section II, paragraph 11a*(2)*. This form should be nonclassified, if possible.

Block 1.a. Report Security Classification: Designate the highest security classification of the report. *(See DoD 5200.00.1-R, Chapters, I, IV, VII, XI, Appendix A).*

Block 1.b. Enter the restricted marking or warning notice of the report *(e.g., CNWDI, RD, NATO).*

Block 2.a. Security Classification Authority: Enter the commonly used markings in accordance with DoD 5200.1-R, Chapter IV, Section 4, paragraph 4-400 and 4-402. Indicate classification authority.

Block 2.b. Declassification/Downgrading Schedule: Indicate specific date or event for declassification or the notation "Originating Agency Determination Required" or "OADR." Also insert *(when applicable)* downgrade to: _____ on _____, *(e.g., "Downgrade to Confidential on 6 July 1983). (See also DoD 5220.22-M, Industrial Security Manual for Safeguarding Classified Information, Appendix II).*

NOTE: Entry must be made in Blocks 2.a. and 2.b. except when the original report is unclassified and has never been upgraded.

Block 3. Distribution/Availability Statement of Report: Insert the statement as it appears on the report. If a limited distribution statement is used, the reason must be one of those given by DoD Directive 5200.20, Distribution Statements on Technical Documents. The Distribution Statement should provide for the broadest distribution possible within limits of security and controlling office limitations.

Block 4. Performing Organization Report Number*(s)*: Enter the unique alphanumeric report number*(s)* assigned by the organization originating or generating the report from its research and whose name appears in Block 6. These numbers should be in accordance with ANSI STD 239.23-74 "American National Standard Technical Report Number." If the Performing Organization is also the Monitoring Agency, enter the report number in Block 4.

Block 5. Monitoring Organization Report Number*(s)*: Enter the unique alphanumeric report number*(s)* assigned by the Monitoring Agency. This should be a number assigned by a Department of Defense or other government agency and should be in accordance with ANSI STD 239.23-74 "American National Standard Technical Report Number." If the Monitoring Agency is the same as the Performing Organization enter the report number in Block 4 and leave Block 5 blank.

Block 6.a. Performing Organization: For in-house reports, enter the name of the performing activity. For reports prepared under contract or grant, enter the contractor or the grantee who generated the report and identify the appropriate corporate division, school, laboratory, etc., of the author.

Block 6.b. Enter the office symbol of the performing organization.

Block 6.c. Enter the address of the performing organization, list city, state and ZIP code.

Block 7.a. Monitoring Organization — Name: This is the agency responsible for administering or monitoring a project, contract, or grant. If the monitor is also the performing organization, leave Block 7.a. blank. In the case of joint sponsorship, the monitoring organization is determined by advanced agreement. It can be either an office, a group, or a committee representing more than one activity, service or agency.

Block 7.b. Enter the address of the monitoring organization. Include city, state and ZIP code.

Block 8.a. Funding *(Sponsoring)* Organization — Name: Enter the full official name of the organization under whose immediate funding the document was generated, whether the work was one in-house or by contract. If the Monitoring Organization is the same as the Funding Organization, leave Block 8.a. blank.

Block 8.b. Enter the office symbol of the Funding *(Sponsoring)* Organization.

Block 8.c. Enter the address of the Funding *(Sponsoring)* Organization. Include city, state and ZIP code.

Block 9. Procurement Instrument Identification Number *(Contract, Grant, or other Funding Instrument)*: For a contractor or grantee report, enter the complete contract or grant number*(s)* under which the work was accomplished. Leave this block blank for in-house reports.

Block 10. Source of Funding *(Program Element, Project, Task Area, and Work Unit Number(s))*: These four data elements relate to the DoD budget structure and provide program and/or administrative identification of the source of support for the work being carried on. Enter the program element, project, task area, work unit number, or their equivalents that identify the principal source of funding for the work required. These codes may be obtained from the applicable DoD forms such as the DD Form 1498 *(Research and Technology Work Unit Summary)* or from the fund citation of the funding instrument. If this information is not available to the authoring activity, these blocks should be filled in by the responsible DoD Official designated in Block 22. If the report is funded from multiple sources, identify only the Program Element and the Project, Task Area and Work Unit Numbers of the principal contributor.

Block 11. Title and Its Security Classification: Enter the title in Block 11 in initial capital letters exactly as it appears on the report. Titles on all classified reports, whether classified or unclassified, must be immediately followed by the security classification of the title enclosed in parentheses. A report with a classified title should be provided with an unclassified version if it is possible to do so without changing the meaning or obscuring the contents of the report. Use specific, meaningful words that describe the content of the report so that when the title is machine-indexed, the words will contribute useful retrieval terms.

If the report is in a foreign language and the title is given in both English and a foreign language, list the foreign language title first, followed by the English title enclosed in parentheses. If part of the text is in English, list the English title first followed by the foreign language title enclosed in parentheses. If the title is given in more than one foreign language, use a title that reflects the language of the text. If both the text and titles are in a foreign language, the title should be translated, if possible, unless the title is also the name of a foreign periodical. Transliterations of often used foreign alphabets *(see Appendix A of MIL-STD-847B)* are available from DTIC in document AD-A080 800.

Block 12. Personal Author*(s)*: Give the complete name*(s)* of the author*(s)* in this order: last name, first name and middle name. In addition, list the affiliation of the authors if it differs from that of the performing organization.

List all authors. If the document is a compilation of papers, it may be more useful to list the authors with the titles of their papers as a contents note in the abstract in Block 19. If appropriate, the names of editors and compilers may be entered in this block.

Block 13.a. Type of Report: Indicate whether the report is summary, final, annual, progress, interim, etc.

Block 13.b. Period of Time Covered: Enter the inclusive dates *(year, month, day)* of the period covered, such as the life of a contract in a final contractor report.

Block 14. Date of Report: Enter the year, month, and day, or the year and the month the report was issued as shown on the cover.

Block 15. Page Count: Enter the total number of pages in the report that contain information, including cover, preface, table of contents, distribution lists, partial pages, etc. A chart in the body of the report is counted even if it is unnumbered.

Block 16. Supplementary Notation: Enter useful information about the report in hand, such as: "Prepared in cooperation with . . ." "Translation at *(or by)* . . ." "Symposium . . ." If there are report numbers for the report which are not noted elsewhere on the form *(such as internal series numbers or participating organization report numbers)* enter in this block.

Block 17. COSATI Codes: This block provides the subject coverage of the report for announcement and distribution purposes. The categories are to be taken from the "COSATI Subject Category List" *(DoD Modified)*, Oct 65, AD-624 000. A copy is available on request to any organization generating reports for the DoD. At least one entry is required as follows:

Field — to indicate subject coverage of report.
Group — to indicate greater subject specificity of information in the report.
Sub-Group — if specificity greater than that shown by Group is required, use further designation as the numbers after the period *(.)* in the Group breakdown. Use only the designation provided by AD-624 000.
Example: The subject "Solid Rocket Motors" is Field 21, Group 08, Subgroup 2 *page 32, AD-624 000).*

Block 18. Subject Terms: These may be descriptors, keywords, posting terms, identifiers, open-ended terms, subject headings, acronyms, code words, or any words or phrases that identify the principal subjects covered in the report, that conform to standard terminology and exact enough to be used as subject index entries. Certain acronyms or "buzz words" may be used if they are recognized by specialists in the field and have a potential for becoming accepted terms. "Laser" and "Reverse Osmosis" were once such terms.

Flexible Parsing

Final Technical Report

For Grant Number: AFOSR-82-0219

on research sponsored by

Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D. C. 20332

and performed by

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Principal Investigator

Philip J. Hayes

Table of Contents

List of Figures

Abstract

When people use language spontaneously, they often do not adhere strictly to commonly accepted standards of grammaticality. The primary objective of this project is to develop flexible computer parsing techniques which can deal with the various kinds of ungrammaticalities that arise, both on the lexical and the phrase level.

The progress towards this goal covered by this report includes:

1. The initial development of the FlexP flexible parser based on pattern-matching techniques.
2. Review of the initial design choices for FlexP in the light of this evaluation, leading to the formulation of the construction-specific approach to parsing, and its preliminary evaluation for applied natural language processing through the experimental parsers CASPAR and DYPAR.
3. Application of the construction-specific approach to flexible parsing to the parsing of an artificial command language in the parser for the COUSIN command interface, a graceful interface for the Unix operating system.
4. Investigation of control structures that would allow the integration of multiple diverse parsing strategies into a single parsing system in an extensible manner.
5. Development of a taxonomy of grammatical deviations and recovery strategies for dealing with them.
6. Design and implementation of an initial version of MULTIPAR, the large-scale robust restricted-domain parser mentioned above that employs multiple construction-specific parsing strategies.
7. Application of the flexible parsing techniques developed under previous parts of the contract to speech input.

1. Research Objectives

1. **development of flexible parsing techniques:** When people use language spontaneously, they often do not adhere strictly to commonly accepted standards of grammaticality. The primary objective of this project was to develop flexible computer parsing techniques which can deal with the various kinds of ungrammaticalities that arise, both on the lexical and the phrase level. The kinds of ungrammaticality we wished to deal with include at the lexical level:

   - misspelt words
   - novel words whose role can be inferred from context
   - erroneous segmentation between words (arising from the omission of spaces, or the inclusion of spurious spaces or punctuation)

- lexical items which are entered in one form and then changed to another

and at the phrase level:

- input which is broken off and then restarted
- interjected words and phrases
- omitted or substituted words and phrases
- fragmentary or otherwise elliptical input
- agreement failure
- idioms

2. **investigation of design choices for flexible parsers**: The design space for parsers is very large. We aimed to develop a set of design choices which will result in parsers well suited to our primary goal. Examples of the design choices we used are:
   - bottom-up rather than top-down parsing, except in certain situations in which top-down prediction is highly constraining
   - use of several different parsing strategies, each tailored to a particular type of construction, and selected between on a dynamic basis
   - provision for the suspension and later resumption of a partial parse at a non-adjacent part of the input string
3. **development of flexible parsing techniques for interfaces to interactive computer systems**: We worked with two types of interface language:
   a. limited-domain natural languages, i.e. languages with the syntax of (possibly a subset of) natural language, but whose semantics are limited to those of the interactive system being interfaced to.
   b. more restrictive artificial languages of the sort currently found in computer interfaces

   Insights gained on these kinds of languages should be transferable to more general natural language.
4. **investigation of formalisms for specifying domain-dependent grammars**: in a convenient way for both of the types of language mentioned above.
5. **applicability of flexible parsing techniques to speech input**: Spoken input is very susceptible to error, both by the speaker and by low-level word recognition techniques. Techniques developed for dealing with errors in typed input are therefore good candidates for use with speech.

As the following description of the accomplishments of this contract shows, these research objectives were largely met.

2. Overview of Accomplishments

The work performed under this contract from its start in 1979 includes several distinct components:

- The initial development of the FlexP flexible parser based on pattern-matching techniques. FlexP was targetted at restricted domain natural languages, and was evaluated in the context of a gracefully interacting interface to an electronic mail system. This work covered a period from the start of the contract in July 1979 to early 1981.

- Review of the initial design choices for FlexP in the light of this evaluation, leading to the formulation of the construction-specific approach to parsing, and its preliminary evaluation for applied natural language processing through the experimental parsers CASPAR and DYPAR. This covered the period from early 1981 to early 1982.

- Application of the construction-specific approach to flexible parsing to the parsing of an artificial command language in the parser for the COUSIN command interface, a graceful interface for the Unix operating system developed largely under funding from the Defense Advanced Research Projects Agency. A good deal of effort went into making this parser robust at the lexical level through spelling-correction and abbreviation expansion of command language keywords and of Unix filenames, including full directory path specifications. This effort started in mid-1981, and continued through mid-1982. It represented a parallel track of development for the construction-specific ideas mentioned above.

- Investigation of control structures that would allow the integration of multiple diverse parsing strategies into a single parsing system in an extensible manner. Control structures of this type are necessary for scaling up the construction-specific approach to a parser with a linguistic coverage complete enough to serve as a realistic natural language front-end to a limited-domain computer system. This work started in early 1982 and ran through late 1983.

- Development of a taxonomy of grammatical deviations and recovery strategies for dealing with them. The taxonomy includes deviations at the lexical and dialogue level, as well as at the sentential level. For each recovery strategy, an attempt was made to assess the computational framework necessary to implement it. This work was also preparatory for a large-scale construction-specific parser. It was done in the latter part of 1983 and early 1984.

- Design and implementation of an initial version of MULTIPAR, our name for the large-scale robust restricted domain parser mentioned above that employs multiple construction-specific parsing strategies. MULTIPAR integrates its multiple strategies through a control structure based on the work mentioned above. MULTIPAR's strategies operate in a highly interpretive manner from definitions of domain entities (objects, actions, and states). This high degree of interpretiveness is important for robustness, since it means the information in the entity definitions can be applied to the input in a variety of ways. This work ran from early in 1984 through late 1985.

- Application of the flexible parsing techniques developed under previous parts of the contract to speech input. This work was done in conjunction with a concurrent DARPA-funded speech project in the Carnegie-Mellon Computer Science Department. The high degree of error in speech input stemming from both speaker errors and recognition

errors made this a fruitful domain of application. This work ran from early 1985 to the end of the contract in June 1986.

Subsequent sections describe each of these pieces of work in greater detail and give references to published descriptions of them.

3. FlexP

3.1. Overview

FlexP is a parser targetted at the restricted-domain natural languages typically used in natural language interfaces. It uses flexible parsing techniques to deal with errors or other ungrammaticalities in the input. It is described in full detail in[11]. The flexible parsing techniques used by FlexP can be divided into two categories: those that operate at the lexical level and those that operate at the phrase level. We discuss them separately below.

The initial application for FlexP is as part of a gracefully interacting user interface that we developed under other support. This application allowed us to use exactly the kind of grammar we are interested in: a natural language grammar restricted to a tightly constrained domain of discourse. We are most interested in grammars of this type because they are the most widely used for natural language communication with machines in such applications as question-answering, data base access, command interfaces, consultation systems, etc.. Some of the parsing techniques we developed take advantage of the constraints afforded by such a grammar.

The graceful interface mentioned above is tool-independent in the sense that it can act as an interface to any tool (functional subsystem) that is described in a certain declarative formalism developed in the interface project. If the tool description is to be complete, it must clearly include an input grammar that defines the way in which the user can specify commands to the tool. We developed a formalism in which it is convenient for a tool builder to specify a grammar suitable for use with our flexible parser.

3.2. Lexical Level Processing

Our work at the lexical level concentrated on spelling correction. We designed and implemented a spelling corrector suitable for use with the phrase level of FlexP. Spelling correction involves comparing an unknown word against a list of known words and finding the known words that are close enough to the unknown word according to some metric to be possible corrected spellings for the unknown word. The assumption being, of course, that the unknown word is unknown because it is misspelt. Our spelling corrector breaks down this process into two steps:

1. find a set of candidates for the corrected spellings,

2. select those of the candidates that are close enough to the unknown word to be corrected spellings of it.

We adopted a two step approach because direct comparison of a misspelt word and a known word according to most reasonable metrics can be quite expensive. The first step in our method allows us to use a much cheaper, though less selective, method to generate an initial set of candidates from the set of all known words, and then to apply the more expensive direct comparison only to the words in this smaller candidate set.

Our direct comparison algorithm is in fact a slightly modified version of the INTERLISP spelling correction algorithm. As such it is of little research interest. It deals with transpositions, and with repeated, substituted, and omitted letters.

The initial selection of candidates is more interesting. Our method selects as candidates those known words which have the longest common substring with the unknown word. Moreover, for this purpose the words are treated as though their letters were arranged in a circle with the first letter following the last. This method makes the selection algorithm insensitive to where in a word a spelling error occurs, whereas a selection based say on common initial substrings would penalise spelling errors near the beginnings of words compared to those occuring near the ends. This method depends on storing all cyclic permutations of all known words in the dictionary. An example will make this clearer. Suppose the word "first" is known to the system. The dictionary would then contain all its cyclic permutations: first%, irst%f, rst%fi, st%fir, t%firs, where a percent sign denotes the end of a word. Suppose that the word to be spelling-corrected is "frist". All of its cyclic permutations are generated, and matched on an initial substring basis against all the items in the permuted dictionary, including those listed above. The best match among those is st%fir which has an initial substring of length three in common with the st%fri permutation of frist. Note that this substring is one longer than the longest common substring of first and frist. This method of selecting initial candidates appears to be worth the approximately five-fold increase in the size of the dictionary that it entails.

Besides spelling correction, we also devoted some attention to incorrect segmentation at the lexical level, so that run together words such as "firstone" can be dealt with. Our approach involves a modification to the second step of the spelling corrector: the step which makes a direct comparison between a known and unknown word. The change is quite simple; if the comparison procedure finds that an initial segment of an unknown word is acceptably close to a known word, then the known word is returned as the correction for the unknown word along with the trailing segment of the

unknown word. This trailing segment can then used as the next lexical item in the input.

The final lexical level topic we looked at was novel words. In general, the spontaneous use of new vocabulary by a user is a very hard research topic. However, we made provision for a special class of novel words: novel proper names. Such names are handled at the phrase level rather than the lexical level, through the inclusion of a special "unknown" word class. Thus, it is the current parsing environment which will decide whether an unknown word is treated as a proper name or a misspelling.

3.3. Phrase Level

The central part of FlexP operates at the phrase level. At this level the parser is intended to parse correctly input that corresponds to a fixed grammar, but also to deal with input that deviates from that grammar in a certain set of ungrammaticalities. These types of ungrammaticality include:

- input which is started one way, and then broken off and restarted
  (e.g. delete the show me all the messages from Smith)

- fragmentary and otherwise elliptical input
  (e.g. "Jim" in response to "Do you mean Jim Smith or Fred Smith?")

- interjected words and phrases
  (e.g. "Display the message dated I think June 17")

- omitted or substituted words or phrases
  (e.g. "Display the message June 17"
  where "dated" or "received after" or some similar phrase is omitted)

- missing or spurious punctuation

To tackle these ungrammaticalities we decided to base our parser on two well-known but relatively little used parsing strategies: bottom-up parsing and pattern-matching.

3.3.1. Bottom-Up Parsing

Our choice of a bottom-up strategy was based on our need to recognize isolated sentence fragments. If an utterance which would normally be considered only a fragment of a complete sentence is to be recognized top-down, there are two approaches to take. First, the grammar can be altered so that the fragment is recognized as a complete utterance in its own right. This is undesirable because it can cause enormous expansion of the grammar, and because it becomes difficult to decide whether a fragment appears in isolation or as part of a larger utterance, especially if the possibility of missing end of sentence markers also exists. The second option is for the parser to infer from the conversational context what grammatical sub-category (or sequence of sub-categories)

the fragment might fit into, and then to do a top-down parse from that sub-category. This strategy is clearly better than the first one, but has two problems; first of predicting all possible sub-categories which might come next, and secondly, of inefficiency if a large number are predicted.

Bottom-up parsing avoids the problem of predicting what sub-categories may occur. If a fragment fitting a given sub-category does occur, it is parsed as such whatever the context. However, if a given input can be parsed as more than one sub-category, the bottom-up approach would have to produce them all, even if only one would be predicted top-down. In a system of limited comprehension, fragmentary recognition is sometimes necessary because not all of an input can be recognized, rather than because of intentional ellipsis. Here, it is probably impossible to make predictions and bottom-up parsing is the only method that is likely to work. Bottom-up strategies are also helpful in recognizing interjections and restarts.

3.3.2. Pattern-Matching

We chose to use a grammar of linear patterns rather than a transition network because pattern-matching meshes well with bottom-up parsing, because it facilitates recognition of utterances with omissions and substitutions, and because it is necessary anyway for the recognition of idiomatic phrases.

A grammar for FlexP is a set of rewrite or production rules whose left hand side is a linear pattern of constituents (lexical or higher level) and whose right hand side defines a result constituent. Elements of the pattern may be labelled optional or allow for repeated matches. We make the assumption that the grammar will be semantic rather than syntactic, with patterns corresponding to idiomatic phrases or to object and event descriptions meaningful in some limited domain, rather than to general syntactic structures.

Linear patterns fit well with bottom-up parsing because they can be indexed by any of their components, and because, once indexed, it is straightforward to confirm whether a pattern matches input already processed in a way consistent with the way the pattern was indexed.

Patterns help with the detection of omissions and substitutions because in either case the relevant pattern can still be indexed by the remaining elements that appear correctly in the input, and thus the pattern as a whole can be recognized even if some of its elements are missing or incorrect. In the case of substitutions, such a technique can actually help focus spelling correction or proper name recognition by isolating the substituted input and the pattern constituent which it should have matched. In effect, this allows the normally bottom-up parsing strategy to go top-down to resolve

such substitutions.

3.3.3. Design Revisions Based on an Initial Implementation

An initial parser implementation based on these ideas helped us clarify some issues in the control of the pattern-matching process. We found that if, on input of a given lexical item, we attempted to match all patterns indexed by the input, a proliferation of unrelated patterns resulted, slowing the parser down, and making it hard to determine a consistent parse for a complete utterance. While the ability to index a pattern from any of its constituents was important for fragmentary or restarted inputs, it tended to generate too many spurious possibilities in more coherent input. Consequently, we produced a considerably refined design for our parser. The key refinement was never to try to match a pattern indexed by an input unless the result of that pattern could be fitted into the partial parse formed up to that point, or unless the pattern was being used to start a new partial parse. With this degree of control, we can reduce the number of active patterns to those that are directly relevant to the current parse. In essence, we introduced a degree of top-down filtering into the pattern-matching procedure, without giving up the responsiveness to input afforded by our essentially bottom-up process. Technically, our refinements make the parser left-corner instead of bottom-up, but since the term, left-corner, is relatively obscure, we prefer to continue describing the parser as bottom-up.

Our experience with the initial implementation also convinced us of the need to allow a partially completed parse to be suspended with the possibility of later continuation. The reasons for this are related to the recognition of interjections, restarts, and implicit terminations. The parsing algorithm works left to right in a breadth-first manner. It maintains a set of partial parses, each of which accounts for the input already processed but not yet accounted for by a completed parse. The parser attempts to incorporate each new input into each of the partial parses. If this is successful, the partial parses are extended and may increase or decrease in number. If no partial parse can be extended, the entire set is saved as a suspended parse.

There are several possible explanations for input mismatch, i.e. the failure of the next input to extend a parse.

- The input could be an implicit termination, i.e. the start of a new top-level utterance, and the previous utterance should be assumed complete.
- The input could be a restart, in which case the active parse should be abandoned and a new parse started from that point.
- The input could be the start of an interjection, in which case the active parse should be temporarily suspended, and a new parse started for the interjection.

It is not possible, in general, to distinguish between these cases at the time the mismatch occurs. If the active parse is not at a possible termination point, then input mismatch cannot indicate implicit termination, but may indicate either restart or interjection. It is necessary to suspend the active parse and wait to see if it is continued at the next input mismatch. On the other hand, if the active parse is at a possible termination point, input mismatch does not rule out interjection or even restart. In this situation, our algorithm tentatively assumes that there has been an implicit termination, but suspends the active parse anyway for subsequent potential continuation.

Note also that the possibility of implicit termination provides justification for the strategy of interpreting each input immediately it is received. If the input signals an implicit termination, then the user may well expect the system to respond immediately to the input thus terminated.

While linear patterns are well suited as the basic grammatical constituents for the types of domain we intend to cover, there are always a few types of construction that do not fit well into this formalism, mainly because of efficiency considerations. Dates, for example, come in so many small perturbations of order (e.g. March 2, 1980; 2 March 1980, 3-2-80, March the second, 1980, etc.) that it would be quite inefficient to parse them through a set of linear patterns, one corresponding to each form. Instead, we have introduced the idea of a special pattern into the parser. A special pattern is not a static pattern at all, but instead a set of special purpose functions which are able deterministically to recognize any of the several forms of their constituent. The set of functions that must be provided for each special pattern is pre-defined by the parser, which calls these functions at the appropriate time. Special patterns thus provide a relatively clean escape from the inefficiencies that can sometimes arise with strict linear pattern matching.

Details of our current parser design and several worked examples of its operation can be found in[11].

3.4. Applications

In designing FlexP we tried to strike a balance between theoretical and practical considerations. Thus, while we believe that many of the techniques embodied by FlexP are of quite general applicability, we have not tried to produce a universal parser. FlexP is, in fact, aimed at languages that can be characterized as domain-restricted subsets of natural language. We are most interested in this type of language because it is the type almost always used to provide natural language access to a functional computer system, and thus is more in need of flexible parsing techniques than other, more general, approaches to language processing. The range of applications of restricted-domain natural language include interfaces to data-bases, command interfaces, question-answering systems, and virtually every other type of application in which natural language has been used in conjunction

with a practically useful underlying computer system.

Our initial application for FlexP is as the parser for a gracefully interacting tool independent user interface that we are developing under other support. Tool independent here means that the system can be used as the interface to any one of an arbitrary number of tools (functional subsystems). The interface gets specific information about each tool from a declarative *tool description*, a database in a formalism we have devised, containing information about the various objects and operations dealt with by the tool. The most important reason for making the interface tool independent is the amount of effort required to make it gracefully interacting, effort which could not be justified for an interface to a single tool. Gracefully interacting here means that the interface appears friendly, supportive, and robust to the user, in contrast to most currently available command interfaces which appear most uncooperative and unfriendly to their users. Naturally, the ability to parse input flexibly is a very important component of graceful interaction. Other aspects of graceful interaction are detailed in [9, 10], and a broader view of the interface project is given in [1].

While restricted natural language is important in a command interface, especially for naive users of that interface, expert users would tend to become frustrated if they had to type all their input in complete English sentences. Crypticness can be dealt with by the flexible techniques we have outlined, so that if the user typed

show messages since June 12

instead of

show me all the messages that arrived since June 12

FlexP could fill in the gaps.

3.5. Language Definition

The tool-independent interface mentioned above can serve as the user interface to any functional subsystem which is specified in a declarative tool description according to a certain formalism. Clearly then, the tool description, besides describing the objects and operations of the tool, must also specify a language in which the user will talk about those objects and operations. For the purposes of our parser, the tool description must define a language in terms of whose grammar the parser will process the user's input. A grammar for FlexP consists of a set of patterns with associated results. However, the formalism used for these rewrite rules is sufficiently complicated to make it inappropriate for direct use in the tool description. In addition, to produce efficient grammars or achieve certain effects, it is necessary to know certain specialized techniques for writing rules that it would be unreasonable to expect the designer of an arbitrary tool to master. Accordingly, we felt the need to provide the tool designer a formalism for the definition of a language which would shield him

from all these details, and instead allow him to describe the language in terms more naturally related to the tool he was designing.

The formalism we designed to meet this need requires the user to specify the language by defining, for each object and operation known to the tool, syntax by which the object or operation may be described. The syntax for an object or operation can be specified as a pattern of words, or as an instance of some syntactic construction, such as noun phrase or transitive verb phrase. For example, the tool description of an electronic mail system might specify the syntax of a message as a noun phrase whose head noun is "message", or "note", or "piece of mail", and whose post modifiers include "from +Sender", "dated +Date", etc.. Here +Sender means anything that can appear in the Sender slot of a message. Note also that the pattern-matching basis of the parser allows the notion of a head noun to be stretched to such phrases as "piece of mail". This specification would allow the user to refer to a message as "the message from Smith", or "a piece of mail dated June 17", and so on. The individual syntax descriptions for each tool object and operation are compiled into a set of rewrite rules suitable for use by FlexP. During the compilation process, this set is supplemented by a number of domain independent constructions that the user is likely to employ. So the user may for instance say "give me more information about x", where x is the name of any object or operation, and this will be interpreted as a request to use the interface's help facility.

4. Construction-Specific Approach to Flexible Parsing

4.1. Overview

After completing the implementation of the FlexP parser as described in Section 3, we evaluated it through use in a gracefully interacting interface developed under other support. This application showed:

- that FlexP was able to parse both grammatical and ungrammatical input according to a simple grammar of pattern-matching rewrite rules,
- that the bottom-up approach of FlexP was helpful in the case of ungrammatical input,
- and that a grammar suitable for use by FlexP could be defined in terms natural to the domain of interaction of the interface.

However, the experimental use of FlexP also made it clear that FlexP had certain problems, largely due to the uniform nature of its grammar. These problems caused FlexP to parse some ungrammatical input inefficiently, and in other cases to generate an unnecessarily large number of alternative interpretations of ungrammatical input.

Based on this evaluation, we reviewed the set of design choices on which FlexP was based to determine if it was possible to resolve the problems that FlexP had without also losing the desirable aspects of its performance. This review led us to replace the design choice of a single parsing strategy based on linear pattern matching with an approach based on multiple parsing strategies, one for each construction class of the language being parsed, with the strategies being selected between on a dynamic basis. This multi-strategy, construction-specific approach was evaluated in a preliminary way through the construction of two small parsers, CASPAR and DYPAR, and was found to show considerable promise.

4.2. Evaluation of FlexP

FlexP was tested extensively in conjunction with a gracefully interacting interface to an electronic mail system[1]. "Gracefully interacting" means that the interface appears friendly, supportive, and robust to its user. In particular, graceful interaction requires the system to tolerate minor input errors and typos, so a flexible parser is an important component of such an interface. While FlexP performed this task adequately, problems of efficiency and of unnecessary ambiguity showed up through this experimentation - examples are given below. The problems that arose are rooted in the incompatibility between the uniform nature of the pattern-matching rewrite rule grammar representation used by FlexP and the kinds of flexible parsing strategies required to deal with the inherently non-uniform nature of some language constructions. In particular:

- Different elements in the pattern of a single grammar rule can serve radically different functions and/or exhibit different ease of recognition. Hence, an efficient parsing strategy should react to their apparent absence, for instance, in quite different ways.

- The representation of a single unified construction at the language level may require several linear patterns at the grammar level, making it impossible to treat that construction with the integrity required for adequate flexible parsing.

The second problem is directly related to FlexP's use of a pattern-matching grammar, but the first would arise with any uniformly represented grammar applied by a uniform parsing strategy.

For our example application, these problems manifested themselves most markedly by the presence of case constructions in the input language. Consider, for example, the following noun phrase with a typical postnominal case frame:

the messages from Smith about ADA pragmas dated later than Saturday.

The phrase has three cases marked by "from", "about", and "dated later than". This type of phrase is actually used in FlexP's current grammar, and the basic pattern used to recognize descriptions of messages is:

```
<?determiner *MessageAdj MessageHead *MessageCase>
```

which says that a message description is an optional (?) determiner, followed by an arbitrary number (*) of message adjectives followed by a message head word (i.e. a word meaning "message"), followed by an arbitrary number of message cases. In the example, "the" is the determiner, there are no message adjectives, "messages" is the message head word, and there are three message cases: "from Smith", "about ADA pragmas", and "dated later than". Because each case has more than one component, each must be recognized by a separate pattern:

```
<%from Person>
<%about Subject>
<%since Date>
```

Here % means anything in the same word class, "dated later than", for instance, is equivalent to "since" for this purpose.

These patterns for message descriptions illustrate the two problems mentioned above: the elements of the case patterns have radically different functions - the first elements are case markers, and the second elements are the actual subconcepts for the case. Also, a single construction at the language level is spread over several patterns in the grammar. This has two undesirable consequences for the parsing process - inefficiency and the generation of unnecessary ambiguities.

First, let us examine how inefficiency arises. Because the parser has no information about the relationship between the cases and the top-level pattern (other than that the results of the case patterns match the last element in the top-level pattern), several powerful, but specialized, strategies for dealing with (regular or irregular) case constructions cannot be employed with a resulting loss of parsing efficiency. For instance, since case indicators are typically much more restricted in range of expression, and therefore much easier to recognize than their corresponding subconcepts, a plausible strategy for a parser that "knows" about case constructions is to scan input for the case indicators, and then parse the associated subconcepts top-down. This strategy is particularly valuable if one of the subconcepts is malformed or of uncertain form, such as the subject case in our example. Neither "ADA" nor "pragmas" is likely to be in the vocabulary of our system, so the only way the end of the subject field can be detected is by the presence of the case indicator "from" which follows it. However, FlexP cannot distinguish case indicators from case fillers - both are just elements in a pattern with exactly the same computational status. Hence it cannot use this strategy and inefficiency results.

The second consequence of the general problems mentioned above is the generation of unnecessary alternative parses. For instance, if an object type can appear in more than one slot of a case frame, and a case indicator for such an object is omitted on input, then a parser dealing with case constructions in an integrated way may be able to resolve the potential ambiguity using

information from what other cases in the case frame are filled, while a uniform strategy would naturally tend to generate all the ambiguous alternatives, and this certainly was the case for FlexP. A specific example arises in the case of:

the messages Jones to Smith

Here, there are two relationships between Persons and Messages - sender and recipient. Since Smith is marked as the recipient, an integrated case strategy can tell that Jones must fill the other relationship, whereas FlexP because of its uniform strategy would get the Smith relation right, but flag an ambiguous relation for Jones.

Examples like these forced us to the conclusion that parsing case constructions efficiently and unambiguously in a flexible manner demands parsing techniques specific to case constructions. In turn, this caused us to review our design decision to use a uniform grammar based on linear patterns, which does not lend itself to such construction-specific parsing techniques. Since similar arguments can be made against other uniform parsing methods, the idea of developing a parser based on a number of different parsing strategies suggested itself.

4.3. Experiments with a Multi-Strategy Approach to Parsing

Parsing using several different construction-specific strategies is a novel approach, so instead of trying to implement a full-scale parser immediately, we decided to try out the ideas in two simplified parsers, CASPAR and DYPAR. CASPAR was designed to show the suitability of construction specific techniques for ungrammatical input, while DYPAR served as a vehicle to investigate the control problems of coordinating several distinct parsing strategies. We describe both of them briefly below. Further details can be found in[12].

CASPAR was designed to use some of the insights about the flexible parsing of case constructions mentioned in the previous section. To keep things as simple as possible, CASPAR was designed only to recognize simple imperative verb phrases, i.e. imperative verbs followed by a sequence of noun phrases possibly marked by prepositions. Examples for an interface to a data base keeping track of course-registration at a university include:

cancel math 247
enroll Jim Campbell in English 324
transfer student 5518 from Economics 101 to Business Administration 111

Such constructions are classic examples of case constructions; the verb or command is the central concept, and the noun phrases or arguments are its cases. Considered as surface cases, the command arguments are either marked by preposition, or unmarked and identified by position, such as the position of direct object in the examples above.

In line with the construction-specific approach, CASPAR was given two quite distinct parsing strategies:

- A strategy to identify the appropriate case frame and activate its case markers and filler-patterns to deal with the rest of the input utterance.

- A strategy to recognize individual constituent case fillers and markers, including the verb, noun phrases in the role of case fillers, and prepositions in the role of case markers.

The first of these strategies is dominant in the sense that it decides where in the input the second, more detailed, recognizer should be applied and what it should try to recognize when it is applied. The second strategy is a simple linear pattern matcher. This is just what is needed for verbs, prepositions, and simple object descriptions such as those in the examples above.

While CASPAR was constructed in as simple a way as possible, the flexibility and robustness that were obtained by providing separate parsing strategies for the two different construction types it recognizes (case and fixed-order linear patterns) is quite striking. The types of grammatical deviation that can be dealt with alone or in combination include:

- Unexpected and unrecognizable (to the system) interjections as in:
  ↑S↑Q↑S enroll if you don't mind student 2476 in I think Economics 247.

- missing case markers:
  enroll Jim Campbell Economics 247.

- out of order cases:
  In Economics 247 Jim Campbell enroll.

- ambiguous cases:
  transfer Jim Campbell Economics 247 English 332.

Moreover, the construction-specific approach of CASPAR allowed it to deal with all these kinds of ambiguity without the inefficiencies and unnecessary ambiguities that arose with FlexP as described in the previous section.

While CASPAR concentrated on dealing with ungrammaticality through construction-specific strategies, our other experimental parser, DYPAR, concentrated on the control problems involved in combining several different parsing strategies. DYPAR has a *Kernel control module* to select the appropriate parsing strategy as a function of the expected input structure, plus three parsing strategies to select among, each with its own grammatical and/or semantic knowledge encodings, and global data structures to share information. These three strategies are:

- **A context-free semantic grammar component**, grouping domain information into hierarchical semantic categories useful in classifying individual words and phrases in the

input language.

- **A partial pattern match component**, represented as pattern-action rules. The patterns may contain individual words, semantic categories (from the semantic grammar), wild cards, optional constituents, register assignment and register reference. This method enables the semantic grammar non-terminal categories to be applied in a much more effective context-sensitive manner than in a pure context-free grammar recognizer.

- **Equivalence transformations** map domain-dependent and domain-independent constructs into canonical form, requiring a fraction of the patterns and semantic categories that would otherwise be needed. If a phrase-structure can be expressed in several different ways, while retaining the same meaning, it is clearly beneficial to first map it into canonical form, rather than being forced to include all possible variants in every context where that constituent could occur.

These three strategies were combined into a single parser through the use of an applicative condition-action cycle in which all matching rules were allowed to fire on each cycle. This allowed these three quite distinct types of strategy to work effectively together.

4.4. Non-Parsing Advantages of a Construction-Specific Approach

Besides showing the promise of a multi-strategy construction-specific approach to parsing for both grammatical and ungrammatical input, our experiments with CASPAR and DYPAR also showed the approach had other advantages, not directly involved in parsing. In particular, the approach made it straightforward to produce localized representations of unavoidable ambiguity, thus enhancing interaction with the user to resolve the ambiguity. In addition, the approach allows task-specific languages, defined in terms natural to the domain, to be used by the parser without a time-consuming compilation phase, thus significantly enhancing the language development process. The remainder of this section expands on these points. Further details can be found in[6]

If a flexible parser being used as part of an interactive system cannot correct ungrammatical input with reasonable certainty, then the system user must be involved in the resolution of the difficulty Experience with our first flexible parser, FlexP, suggested that the way requests for clarification in such situations are phrased makes a big difference in the ease and accuracy with which the user can correct his errors, and that the user is helped most by a request focusing as tightly as possible on the exact source and nature of the difficulty. As the following examples show, this type of focused interaction was very difficult to arrange with FlexP, but was straightforward using the construction-specific approach of CASPAR.

The following input is typical for the electronic mail system interface[1] with which FlexP was extensively used:

the messages from Fred Smith that arrived after Jon 5

The fact that this is not a complete sentence in FlexP's grammar causes no problem. The only real difficulty comes from "Jon", which should presumably be either "Jun" or "Jan". FlexP's spelling corrector can come to the same conclusion, so the output contains two complete parses which are passed onto the next stage of the mail system interface. The first of these parses looks like:

```
[DescriptionOf: Message
 Sender: [DescriptionOf: Person
          FirstName: fred
          Surname: smith
         ]
 AfterDate: [DescriptionOf: Date
             Month: january
             DayOfMonth: 5
            ]
]
```

This schematized property list style of representation should be interpreted in the obvious way.

If the next stage of the interface has access to other contextual information which allows it conclude that one or other of these parses was what was intended, then it can proceed to fulfill the user's request. Otherwise it has little choice but to ask a question involving paraphrases of each of the ambiguous interpretations, such as:

Do you mean:

1. the messages from Fred Smith that arrived after January 5
2. the messages from Fred Smith that arrived after June 5

Because it is not focused on the source of the error, this question gives the user very little help in seeing where the problem with his input actually lies.

One straightforward solution to the problem is to augment the output language with a special ambiguity representation. The output from our example might look like:

```
[DescriptionOf: Message
 Sender: [DescriptionOf: Person
          FirstName: fred
          Surname: smith
         ]
 AfterDate: [DescriptionOf: Date
             Month: [DescriptionOf: AmbiguitySet
                     Choices: (january june)
                    ]
             DayOfMonth: 5
            ]
]
```

This representation is exactly like the one above except that the Month slot is filled by an AmbiguitySet record. This record allows the ambiguity between january and june to be confined to the month slot where it belongs rather than expanding to an ambiguity of the entire input as in the first

approach we discussed. By expressing the ambiguity set as a disjunction, it would be straightforward to generate from this representation a much more focused request for clarification such as:

Do you mean the messages from Fred Smith that arrived after **January** or **June** 5?

However, this approach only works if the ambiguity corresponds to an entire slot filler. Suppose, for example, that instead of mistyping the month, the user omitted or so completely garbled the preposition "from" that the parser effectively saw:

the messages Fred Smith that arrived after Jan 5

In the grammar used by FlexP for this particular application, the connection between Fred Smith and the message could have been expressed only by "from", "to", or "copied to" (or synonyms thereof). To represent the ambiguity, FlexP generates three complete parses isomorphic to the first output example above, except that Sender is replaced by Recipient and CC in the second and third parses respectively. Again, this form of representation does not allow the system to ask a focused question about the source of the ambiguity. The previous solution is not applicable because *the ambiguity lies in the structure of the parser output rather than at one of its terminal nodes*. Using a case notation, it is not permissible to put an "AmbiguitySet" in place of one of the deep case markers. To localize such ambiguities and avoid duplicate representation of unambiguous parts of the input, it is necessary to employ a representation like the one we designed for CASPAR:

```
[DescriptionOf: Message
 AmbiguousSlots: (
                  [PossibleSlots: (Sender Recipient CC)
                   SlotFiller: [DescriptionOf: Person
                                FirstName: fred
                                Surname: smith
                               ]
                  ]
                 )
 AfterDate: [DescriptionOf: Date
             Month: january
             DayOfMonth: 5
            ]
]
```

This example CASPAR output is similar to the two given previously, but instead of having a Sender slot, it has an AmbiguousSlots slot. The filler of this slot is a list of records, each of which specifies a SlotFiller and a list of PossibleSlots. The SlotFiller is a structure that would normally be the filler of a slot in the top-level description (of a message in this case), but the parser has been unable to determine exactly which higher-level slot it should fit into; the possibilities are given in PossibleSlots. With this representation, it is now straightforward to construct a directed question such as:

Do you mean the messages **from**, **to**, or **copied to** Fred Smith that arrived after January 5?

The adoption of such representations for ambiguity has profound implications for the parsing

strategies employed by any parser that tries to produce them. For each type of construction that such a parser can encounter, the parser must "know" about all the structural ambiguities that the construction can give rise to, and must be prepared to detect and encode appropriately such ambiguities when they arise. Construction-specific parsing techniques as used in CASPAR fit this requirement perfectly. Each construction-specific parsing strategy can encode detailed information about the types of structural ambiguity possible with that construction and incorporate the specific information necessary to detect and represent these ambiguities.

This section concludes with a brief note about language definition. As we described in Section 3, FlexP had a language definition facility which allowed the designer of a task-specific language to define the language without having to know the exact details of FlexP's grammar formalism. This made it much easier to define such languages, but the facility turned out to be inconvenient to use in practice because of the time-consuming compilation phase necessary to transform the language definition in domain terms into FlexP's pattern-match rule formalism. This was particularly inconvenient when a large number of relatively minor changes need to be made, as is normal during language development.

For CASPAR, we implemented a similar language definition facility, but with one important difference - instead of compiling the language definitions into a different formalism, we designed CASPAR to interpret them directly. This made the language designer's job much easier, by letting him make the many small changes that are always necessary in the course of developing a language, without requiring him to go though a time-consuming compilation for each incremental change. The reason that it was possible to do this with CASPAR, and not with FlexP, relates directly to the construction-specific approach that CASPAR embodies. Since the constructions CASPAR deals with correspond directly to those that are natural to the domain, direct interpretation of a language representation designed around these constructions was straightforward for CASPAR.

5. Cousin Command Language Parser

5.1. Overview

At an early stage in the development of the multi-strategy, construction-specific approach to parsing restricted domain natural language described in Section 4, it became apparent to us that a similar approach could be used to parse artificial command languages as well. Accordingly, from mid 1981, we began to develop a flexible parser based on this approach for the Cousin interface to the Unix operating system, which we are developing under funding from the Defense Advanced Research

Projects Agency, and which uses an extended version of the standard artificial Unix command language for input. This effort, which resulted in a complete parser in mid 1982, constituted a development track for the construction-specific approach parallel to that represented by CASPAR and DYPAR and their successor. The two tracks, however, are not completely independent, since several of the specific techniques developed for CASPAR also turned out to be useful for the COUSIN parser, as the description in the remainder of this section will show. More details of the COUSIN system and its parser can be found in[13]. The following two subsections describe the command and lexical levels of the COUSIN parser separately.

5.2. Command Level

The command language for COUSIN is the Unix language, minus the constructions at a level higher than single commands, but supplemented by other language features that make it easier for the user to specify commands. The standard Unix format for command lines is:

<command-name> <options> <arguments>

where <options> is a possibly empty sequence of flags, single characters preceded by dashes, and option markers, also single characters preceded by dashes which identify the next input token as an optional parameter. The <arguments> are a fixed order sequence of parameters to the command that are not identified by any markers, although they may in some cases be optional. An example is:

cc -w -O -o bar foo.c fum.c

which is a call to the C language compiler (cc) with options "w" (suppression of warning diagnostics) and "O" (object code improvement), a flagged option "o" (which writes output to the file named, "bar"), and two arguments foo.c and fum.c, the files to be compiled. Conceptually, cc actually has one argument, the file to be compiled, which may be filled an arbitrary number of times; this type of argument is called a *multiple argument*. A command with two arguments is "cp", which copies a list of files, its first argument, into a directory, its second argument, as in:

cp file1 file2 dir

COUSIN makes two extensions to the standard Unix language: the addition of explicit markers for command arguments as a supplement to the present system of purely positional specification, and the addition of full word flags and markers for options as a supplement to the present system of single characters preceded by dashes. So the above examples could be written for instance as:

cc -O no-warnings foo.c fum.c output-to bar
cp onto dir from file1 file2

When whole-word markers are used, the ordering restrictions of standard Unix are relaxed. Note that this extension makes the language similar in many ways to the kind of language handled by CASPAR - a command verb followed by a set of marked cases. The major differences are that some case markers stand by themselves and have no fillers, and that the Unix positional syntax is still included in

the language. This similarity is exploited by some of the flexible parsing techniques described below.

The multi-strategy construction-specific parsing algorithm that we have so far developed for this language is as follows:

1. **Command Identification:** In much the same way as CASPAR finds the verb of its sentence, the COUSIN parser determines which command is being invoked, and locates the syntax description - positional and case information - for the command.

2. **Standard Unix parsing:** Using this syntax information the remaining part of the command line is parsed as though it conformed to the standard Unix syntax for that command, taking only Unix style options and positionally specified arguments into account. If this step is successful, parsing is complete, and no attempt is made to use the case style syntax. This ensures that correct Unix commands which happen by coincidence to use case marker keywords will be recognized correctly.

3. **Extended Unix parsing:** If the standard parse is unsuccessful in any way, the next step is to parse the line according to the extended syntax. The procedure here is the CASPAR case marker scanning algorithm, modified only to deal with case markers with no corresponding case fillers; i.e., a scan is made for any argument marker keywords, or any option keywords, and the arguments and options thus flagged are extracted.

4. **Flexible Unix parsing:** Otherwise, if any of the input string is still not accounted for after this step, a more flexible algorithm is applied. This algorithm is designed to deal with situations in which the user has:

    - used a mixture of marker and positional notation
    - misspelt input tokens, either arguments or markers (see below on lexical level processing for details on the spelling correction techniques used).
    - used positional notation in the standard Unix style, but has got the arguments out of order
    - omitted one or more required arguments
    - used standard dash notation with single character flags and markers for options, but has omitted the dash or put the option string other than at the beginning of the input.

    Two basic techniques are involved in this flexible style of parsing: scanning for misspelt markers and options, and comparing permutations of the arguments against the input tokens. The first of these is a CASPAR style marker scan, with the possible targets for correct spellings restricted to be markers of the arguments not yet filled. The second technique is specific to the positional style of construction allowed by Unix, and is kept combinatorially tractable by the fact that no Unix command has more than three arguments.

An example will illustrate how this algorithm operates. Suppose, for instance that the user types:

cp onto dir form fil2 file3

when he really intended to type:

cp onto dir from file2 file3

Assume that "dir" is a valid directory name, "file2", "file3" are valid file names, but "onto", "form", and "from" are not valid files or directories. The command cp has two arguments, SOURCE and DESTINATION. SOURCE is a multiple argument of readable files. DESTINATION is an ordinary argument of either a writable directory, or a creatable file (which may or may not already exist). There is an additional restriction that if DESTINATION is a file, SOURCE may contain only one file. The default order is SOURCE DESTINATION.

Standard Unix syntax does not work, so extended Unix syntax is tried. The marker scan comes up with "onto", and "dir" is recognized as a proper DESTINATION, and there are just three remaining arguments which could be assigned to SOURCE, but "form" and "fil2" have failed matches with SOURCE, so extended Unix syntax does not work, and flexible parsing must be tried. Note that if "form" and "fil2" were suitable files for SOURCE, there would have been no need to employ the extra flexibility. The first flexible step is to scan for misspelt markers from left to right. Extended Unix syntax has already accounted for "onto" and "dir", so the scan starts from "form", which is of course corrected to "from". Since "from" is the marker for SOURCE, "fil2" is required to fill the SOURCE argument, and since "file3" satisfies the restrictions for SOURCE, and since SOURCE is a multiple argument, "file3" also is taken into the SOURCE argument. Since "fil2" is required to go into SOURCE the fact that it fails the restrictions on the argument trigger an immediate attempt to spelling correct it. This attempt succeeds, and the parse is correct and complete, without it being necessary to invoke the second permutation phase of flexibility.

This parsing algorithm has proved efficient in the recognition of grammatical input, and robust in its handling of ungrammatical input. In addition, its construction-specific character has made it easy to produce localized representations of ambiguity in its output, which are, as described in Section 4, very important for graceful interaction with the user to resolve the ambiguity.

5.3. Lexical Level Processing

Statistically, the greatest concentration of errors in typed input are typos or other errors in single words. Lexical level processing is thus very important for the performance of flexible parsers in general, and for the COUSIN parser in particular.

The lexical level of the COUSIN parser raises several specialized problems, some of them common to artificial command languages in general, and others related to the particular command language

involved. First, there are many proper names, such as the names of files, which cannot be in the vocabulary of the parser, but must be verified by asking the underlying system whether such files exist, or in the case of commands that create new files, must be checked for validity as file names. As a part of the COUSIN parser, we implemented such routines to interrogate the Unix operating system. We also integrated the spelling correction component of the parser with this interrogation. This spelling correction procedure operated by obtaining a list of all the files in the current directory and spelling correcting unknown tokens that did not turn out to be correct file names against this list. The algorithm also took account of the access rights that were required for the command argument for which the token was a candidate filler, preferring those files with correct access rights over those without the correct rights. A similar spelling correcting interface to the file system would be needed for any error correcting parser for an operating system command interface.

A lexical problem more specific to the Unix system arises because in Unix files are often specified by paths through a tree of directories in addition to the actual file name, as in /usr/pjh/text/report.txt, where usr, pjh, and text are directory names. When spelling errors occur in such paths, they need to be resolved by spelling-correcting each element of the path separately, rather than trying to correct the specification as a whole. We included a special component in the COUSIN parser to deal with this. The component interacted with the file spelling corrector to resolve errors in such paths by maintaining a search tree of routes from the root of the path to its tip. At each step in the search, the tree was expanded in a breadth-first manner by looking up the next path element name in each directory currently at the frontier of the search tree. If the element was found in any of the directories, the search tree was extended only for those directories, otherwise, spelling correction was attempted on the element in all of the directories, and the search tree expanded for each correction that was found, even if there were several in a single directory. In this way, if a spelling correction of one element of the path turns out to be ambiguous, subsequent path elements can be used to select between the ambiguities.

We also provided the users of COUSIN with the convenience of initial substring abbreviation wherever it does not result in ambiguity. Thus "copy report.txt onto backup.txt" could be abbreviated to "co r ont bac" if this was sufficient to avoid ambiguity. Problems, however, arise in spelling correcting such initial abbreviations, especially when they are very short; a single letter abbreviation can be changed into any other single letter by one spelling correction step. We therefore did not attempt spelling correction on one or two character tokens. Note that the possibility of a combination of spelling correction and initial substring abbreviation had an important effect on our spelling correction algorithm. We treated initial substring abbreviation as a mild form of spelling error and organized the left-to-right progress of our spelling correction algorithm so that running out of input

word before getting to the end of the target word was counted as though the target word had been recognized, so long as no spelling errors had been encountered or there were at least three characters in the input word.

In command languages in general and in Unix in particular, files are often specified through the use of *wildcard* symbols. For instance "*", which means any sequence of characters, can be used to specify all the files beginning with "r" and ending with ".txt" through the lexical unit "r*.txt". We incorporated interpretation of such wildcard characters into the lexical component of the COUSIN parser, but have not yet been able to combine such interpretation with spelling correction, so that wildcarded file descriptions are not spelling corrected if they do not match any existing files. An example of the kinds of problem we ran into is that while "r*.txt" may be a reasonable spelling correction for "r*.xt", "*r.xt" would probably not be, since the latter alternative would produce files that were radically different from those specified by the erroneous input, while the former correction would produce only files that were simple spelling corrections of those that the erroneous input could have expanded to.

The kinds of lexical processing described in this section are highly specific and depend on particular kinds of lexical items found only in interactive command languages and in some cases only in the Unix command language. Nevertheless, these particular pieces of lexical processing turned out to be crucial for adequate flexible parsing in the COUSIN interface. This suggests that, in general, the construction-specific approach to parsing should extend down to the lexical level, so that a whole range of lexical correction techniques, each specific to a particular class of lexical unit can be invoked when lexical level errors occur. Indeed, it is difficult to see in several of the preceding examples how adequate corrections could be made without highly specific and specialized correction techniques. The lesson to be learned here seems to be that we should expect that any domain in which we attempt error correcting parsing will give rise to such idiosyncratic lexical level problems, and that we should be prepared to develop correspondingly specific techniques to take care of them.

6. Control Structures for Multiple Parsing Strategies

As described in Section 4, the two simple parsers, CASPAR and DYPAR, we constructed to experiment with the use of different parsing strategies for different construction types showed that the approach had considerable promise. CASPAR, for instance, showed the significant advantages that can be obtained in parsing ill-formed case constructions by employing a construction-specific strategy which treats case markers and case fillers differently (see[3] for further details). In addition to finding the construction-specific strategies valuable for flexible parsing, we also found that we could

get superior performance in parsing grammatical input by using construction-specific techniques. Using a construction-specific approach throughout also allowed us to integrate the parsing of grammatical and extra-grammatical input.

Even though CASPAR and DYPAR were such positive experiences, they were still very simple parsers with very limited ranges of applicability. In particular, they were much too limited to provide adequate linguistic coverage for a non-toy natural language interface, even for a typical restricted domain. Therefore, after CASPAR and DYPAR were completed around the end of 1981, we began to look at ways of producing a parser, based on the same principles, but with a more adequate linguistic coverage. We called the parser that we were aiming at, MULTIPAR. MULTIPAR's eventual implementation is described in Section 8. To obtain the extended coverage while remaining based on the same construction-specific ideas, it had to deal with many more construction types than CASPAR or DYPAR, and therefore had to incorporate many more strategies than the two and three respectively in those parsers.

More specifically, we established the following design goals for MULTIPAR:

- Integration of a large number of highly specific and specialized parsing strategies. There may well be several strategies applicable in any given situation.
- Easy incorporation of new strategies into the existing set.
- Ability to parse bottom-up from the best information available. It is never possible to rely absolutely on any specific piece or feature of a construction being correct.
- As much top-down control as possible. While bottom-up parsing is necessary to form an initial hypothesis about what the structure of an input may be, it is inefficient once that hypothesis has been formed.
- Clean separation between domain semantics and parsing strategies. This is most important because of our intention to apply MULTIPAR to a significant number of different domains (see the subsection on language definition in the proposed research section).

The second and third of these goals were derived from our earlier experience with the FlexP parser which demonstrated the importance of mixing top-down and bottom-up parsing in dealing with ungrammatical input.

The key issue in meeting these design goals was the coordination of the various strategies that would be involved in a way that could accommodate the top-down and bottom-up directionalities that we required. DYPAR and CASPAR used two and three different parsing strategies respectively, and coordination between these strategies was simple and "hard-wired" directly into the control structure

of the parsers themselves. The much larger number of strategies needed to provide adequate linguistic coverage and the need to make the addition of new strategies easy preclude this "hard-wired" approach for MULTIPAR. Therefore, we expended significant effort between early 1982 and late 1983 in developing a control structure which allows large numbers of strategies to cooperate on and share information about the parsing of a given input.

The control structure we developed as a result of this work involves the following three kinds of object:

- **tasks**: A task represents the goal of recognizing as much as possible of a given subsequence of the input as a certain kind of grammatically specified object (e.g. a task might be to recognize as much as possible between the second and seventh words of "is the price of a display terminal more than a hardcopy terminal" as a <comparable-object>, where <comparable-object> was a grammatical subcategory). Such tasks may specify that the recognition is to be left or right anchored if the whole subsequence cannot be parsed as the desired object. MULTIPAR is driven at the top-level by a task to recognize the whole of an input line as a grammatical super-category, which includes all complete sentences as well as individual objects, and anything else the system being interfaced to is prepared to interpret in isolation.

- **strategies**: A strategy is a method for recognizing a given grammatical constituent. There may be several strategies applicable to any given grammatical category, and a given strategy may apply to more than one type of constituent. Strategies are indexed by grammatical category. Each strategy has a simple initial test based on pattern matching to check applicability to a specific task (i.e. recognizing a given constituent in a given context with possible left or right anchoring), plus a more complicated procedural test of applicability to be applied if the pattern match succeeds. Each strategy has an indication of the amount of grammatical deviation it is designed to cope with, which will correspond roughly to the amount of effort needed to apply it. Strategies may also be limited to left or right anchored recognition.

- **hypotheses**: An hypothesis is the result of applying a specific strategy to a specific task and constitutes the result of the parsing attempt, thus specified. Hypotheses are recorded globally in a blackboard-like[4] structure. Both successful and unsuccessful attempts are thus recorded, and constitute a way of sharing effort between different strategies. The successful ones are analogous to (partial) parse trees.

These three types of structure work together as follows:

1. The top-level task is set up as described above.

2. Given a task, all strategies whose indexing identifies them as suitable for that task are identified, and grouped according to degree of grammatical deviation handled.

3. The strategies are applied in order of ascending ungrammaticality until one succeeds. All strategies for a given level of ungrammaticality are applied (conceptually) in parallel.

4. Application of a strategy means first checking for a precomputed result in the global blackboard of hypotheses, then applying the pattern-match test, then the procedural test, and then if that succeeds, the body of the strategy.

5. The body of a strategy can set up new tasks, and the strategy as a whole succeeds if the sub-tasks succeed.

6. A task succeeds if one or more of its strategies succeed.

The following examples (drawn from the domain of a computer salesperson's assistant) show how this control structure is intended to operate.

A very simple strategy is:

StrategyName: comparative-sentence

Recognizes: <complete-sentence>

Pattern: [<be> $X <comparative> $Y]

Body: set up subtasks of recognizing input segments represented by X and Y as <comparable-object>s.

Most of the work in this strategy is done by the simple pattern-matching rule which is its initial test. To see how it might operate consider the input

Is the price of a display terminal more than $100

The strategy would be applicable, and would isolate "the price of a display terminal" as X and "$100" as Y. The two subtasks of parsing X and Y as <comparable-object>s would then be established, with the first being parsed by a strategy which recognized constructions of the "<attribute> of <object>" type, and the second which recognized strings beginning with a dollar sign and followed by digits as sums of money. The strategy would also check that the two quantities were comparable before reporting success, trying coercion at a more flexible stage, and thus making sense of "is a display terminal more than $100".

A more complicated strategy is:

StrategyName: imperative-caseframe

Recognizes: <complete-sentence>

Pattern: [<action-word> $X] (a more flexible version would not be left anchored)

Body: Obtain the case frame of the action word. Scan the input segment represented by X for case markers from that case frame. This divides X up into a number of segments separated by case markers. Set up tasks to recognize objects of the

type indicated by the preceding marker for each segment, making allowance for direct and indirect objects.

This second strategy is very similar to the dominant strategy of the CASPAR parser. An example input to which it would be applicable is:

replace the display terminal with a teletype

Here "replace" is the action word and "with" is a marker from its case frame. This isolates "the display terminal" and "a teletype" which can be parsed as objects of the appropriate type, in this case <component-set>s.

For an example of flexibility in response to ungrammatical input, suppose the case marker "with" is missing, so that the two component phrases cannot be isolated. The strategy then sets up tasks to recognize each of the missing case fillers in the string that it cannot split up. Since the strategies always operate to recognize as much of the given subsequence as possible as the requested category, but will ignore parts that they cannot deal with, the attempt to recognize (in left-anchored mode) a component in "the display terminal a teletype", will recognize "the display terminal", fail to recognize "a teletype", but isolate it, thus leading to its recognition on the second attempt to parse still unrecognized strings as the fillers of unfilled case frame slots.

Of course, there is no guarantee, given the many roles that individual prepositions fill, that a case marker that is found is really a case marker for the given case frame, as in:

replace the display terminal with a teletype with a paper-tape reader

Here both "with"s are found, leading to two different ways in which the input can be split up for further parsing. The correct reading is finally preferred because it accounts for more of the input, the strong domain constraints making it easy for the parser to refuse to accept "the display terminal with a teletype" as a <component-set>.

7. Taxonomy of Grammatical Deviations

In addition to appropriate control structures, a large-scale multi-strategy parser needs a variety of parsing strategies. Given the goals of the project, we were particularly interested in strategies designed to recover from ungrammatical input. Accordingly, we undertook in the latter part of 1983 an extensive examination of the various types of grammatical deviation that can occur for restricted-domain interfaces and the parsing strategies necessary to recover from them. The resulting taxonomy is presented in[2], and will not be detailed here.

For each of the recovery strategies examined, an attempt was made to assess the computational framework necessary to implement it. This involved examining how the strategy might be

implemented in each of three standard approaches to parsing, based on transition nets, pattern matching, and case frame instantiation respectively. The first conclusion was that case frame instantiation was generally superior to the other two approaches as a basis for recovery strategies. Further examination of the recovery strategies led us to formulate a set of four characteristics that a parsing process needs to have to form a good basis for recovery strategies. We concluded that case frame instantiation outperformed transition networks and pattern matching because it satisfied the characteristics more closely. The characteristics are:

- The parsing process should be as interpretive as possible. There was a clear need for a parsing process to "stand back" and look at a broad picture of the set of expectations (or grammar) it is applying to the input when an ungrammaticality arises. The more interpretive a parser is, the better able it is to do this. A highly interpretive parser is also better able to apply its expectations to the input in more than one way, which may be crucial if the standard way does not work in the face of an ungrammaticality.

- The parsing process should make it easy to apply semantic information. Semantic information is often very important in resolving ungrammaticality, and the semantic constraints available in restricted domain languages are usually very powerful.

- The parsing process should be able to take advantage of non-uniformity in language. Recovery can be much more efficient and reliable if a parser is able to make use of variations in ease of recognition or discriminating power between different constituents of a construction. This kind of "opportunism" can be built into recovery strategies.

- The parsing process should be capable of operating top-down as well as bottom-up. There are recovery strategies where each of these modes is essential.

This analysis was highly supportive of our preference for a multi-strategy, construction-specific approach. While case frame instantiation is the uniform approach to parsing that fits the above characteristics best, a multi-strategy approach (in which case frame instantiation is heavily represented) fits them even better because:

- The required high degree of interpretiveness can be obtained by having several different strategies apply the same grammatical information to the input in several different ways.

- Strategies can be written to take advantage of particular aspects of non-uniformity among the constituents of individual construction types.

- Some strategies can operate top-down and others bottom up.

8. MULTIPAR

This subsection describes the implementation of MULTIPAR, our intitial version of a large-scale restricted-domain parser using multiple construction-specific parsing strategies. MULTIPAR was implemented and refined into the form described here during the period early 1984 through late 1985. It can be seen as the culmination of the work started with FlexP, and continued through CASPAR, DYPAR, and the other theoretical investigations described above. It represents an important step forward in the parsing of ungrammatical input for restricted domains. We describe first MULTIPAR's control structure, then the domain entity descriptions on which MULTIPAR's parsing strategies are based, and finally show by means of a worked example, just how MULTIPAR operates on some ungrammatical input. Further details can be found in[5] and[14].

8.1. MULTIPAR control structure

Like most natural language parsers, MULTIPAR operates by searching through a space of possible parses for a given input. The size of the search space depends on the number of local ambiguities that are encountered during the parsing process. Because MULTIPAR is expected to parse ungrammatical input, it is typically confronted with a search space that is much larger than that explored by conventional parsers. Unlike the conventional systems, MULTIPAR cannot simply reject a partial parse when a grammaticality constraint is violated. Instead, various recovery techniques are applied. This subsection describes how MULTIPAR controls the exploration of this large search space. The techniques used can be seen as further developments, based on intervening experience, of the design described in Section 6.

If the best parse is to be found in a timely manner, the exploration of alternative paths in the search space must be carefully controlled. For example, spelling correction should be tried before hypothesizing a missing word, and hypothesizing one word is preferable to hypothesizing five. Furthermore, if a strategy fails to find a correct parse for some input, it does not mean that the recovery actions should be immediately invoked. It may be that some other strategy will find a grammatical parse; that strategy should be given its chance before any recovery actions are attempted.

To control the exploration of the search space, competing alternatives within a strategy are explicitly specified using SPLIT statements. When a SPLIT statement is encountered the computation divides into parallel branches; each branch has a *flexibility increment* indicating the degree of ungrammaticality added by the associated action. A separate partial parse is generated along each branch, and each computation proceeds from the SPLIT statement independently of the others.

```
(Split (+0 actionA)
       (+1 actionB)
       (+3 actionC)
       ....)
```

Execution of the statement above produces a three-way branch in the search tree. Alternative A does not result in a gain in flexibility (implying grammaticality) of the associated parse. Alternatives B and C will result in gains of one and three respectively.

The flexibility level of a partial parse is the sum of all flexibility increments used to achieve the parse. The MULTIPAR control structure guarantees that parses are attempted in strict flexibility order and generates all parses at the lowest flexibility level at which any parse can be found. In particular, if a grammatical parse can be found, then all and only grammatical parses will be generated.[1] In the example above, alternative A requires no increase in flexibility and therefore it immediately continues execution. Because alternatives B and C are associated with positive flexibility increments, they are put on an agenda for later consideration. If no parse succeeds at the current flexibility level, then the alternatives at the +1 level are attempted (including alternative B). If no parse is found at the +1 level, then the +2 level alternatives are tried, and later, if necessary, the +3 alternatives (including alternative C).

8.2. MULTIPAR entities

MULTIPAR is an entity-oriented parser. As described in[7], entity-oriented parsing is an approach to restricted domain natural language processing in which the parser is driven by a set of definitions of domain entities (objects, operations, and states). The entities are defined at a high level of abstraction, primarily in terms of other component entities. This approach to parsing is well adapted to dealing with ungrammatical input; it allows a parser to interpret the abstract entity definitions in a variety of ways so that it can look for the entity components in places other than the syntactically correct ones.

An example of an entity definition used by MULTIPAR is:

---

[1] The non-determinism provided by the SPLIT statement is also exploited for handling normal ambiguities. When a strategy finds that a parse is ambiguous, a separate branch is created for each alternative parse. +0 flexibility increments are used in this case.

```
(EntityName     MoveCommand
 SemanticCases  (
        Object      (FileObjDesc or DirectoryObjDesc)
        Source      (DirectoryObjDesc or LogicalDeviceObjDesc)
        Destination (FileObjDesc or DirectoryObjDesc)
        Location    (DirectoryObjDesc or LogicalDeviceObjDesc))
 Constraints    (
        (Destination FileObjDesc > Object FileObjDesc)
        (Object DirectoryObjDesc > Destination DirectoryObjDesc)
        (Object DirectoryObjDesc > Source LogicalDeviceObjDesc)
        (Object DirectoryObjDesc > Location LogicalDeviceObjDesc)
        (Required Object Destination))
 SurfaceForms   (
        (SFName Icf-Canonical
         Head           <movehead>
         DirectObject   Object
         Cases          (
                (Preposition <sourcepreps>
                 Bind Source)
                (Preposition <destpreps>
                 Bind Destination)
                (Preposition <locpreps>
                 Bind  Location))))
 InstanceTemplate (
        Action  'MOVE
        Deviations Deviations
        Source (
                IsA (IsA in Object)
                Name (Name in Object)
                Extension (Extension in Object)
                Directory (Directory in Object or
                           Directory in Source or
                           Directory in Location)
                LogicalDevice (LogicalDevice in Object or
                               LogicalDevice in Source or
                               LogicalDevice in Location)
                ObjDesc  (Description in Object)
                SourceDesc (Description in Source)
                LocDesc (Description in Location))
        Destination (
                IsA (IsA in Destination)
                Name (Name in Object or Name in Destination)
                Extension (Extension in Object or Extension in Destination)
                Directory (Directory in Destination or
                           Directory in Location)
                LogicalDevice (LogicalDevice in Destination or
                               LogicalDevice in Location)
                DestDesc (Description in Destination)
                LocDesc (Description in Location))))
```

This defines the move command for an operating system interface. Like all entity definitions, in addition to its name it has four main parts:

- **SemanticCases**: this defines the basic structure of the entity in terms of the other

entities that are its components. MoveCommand, for instance, has an object (the file[2] or directory to be moved), a source (the directory or logical device to move it from), a destination (the file or directory to move it to), and a location (the directory or logical device in the context of which the move takes place). These semantic cases (or case frame) do not tell MULTIPAR how to recognize the entity in an input utterance, they just define what other entities need to or may be found in the input as part of finding the entity whose definition it is. The general format is a list of case names and filler types.

- **Constraints**: The constraints specify both relations that are required to obtain between cases and predicates that obtain on individual cases for any specific instance of an entity. The first constraint for MoveCommand says that if the Destination case is filled by a file, then the Object case must also be filled by a file. The following three constraints have analogous interpretations. The final constraint says that the Object and Destination cases are required to be present for any instance of MoveCommand. MULTIPAR enforces constraints of both types insofar as it can. In other words, it prefers parses in which the constraints are satisfied, but will accept (as a grammatical deviation) inputs in which the constraints are violated.

- **SurfaceForms**: This component of an entity definition tells MULTIPAR how an entity can be described in English, i.e., it tells MULTIPAR where in the input to find fillers for the SemanticCases, plus possibly some words identifying the entity itself (e.g. "move" or "transfer" in our example). The information about where to look for the SemanticCases is implicit in the specific parsing strategies associated with the SurfaceForm name (Icf-Canonical, or imperative case frame canonical, above). [Of course, a strategy can look in more than one place, depending on the amount of grammatical deviation it is set up to deal with.] Although there is currently a 1-to-1 mapping between SurfaceForm and strategy, it is not necessary that this be the case. A strategy can know how to parse more than one surface form and/or a SurfaceForm can be used by more than one strategy.

  The SFName is the only attribute common to all SurfaceForms. The other attributes are specific to particular surface forms. In the above example, the Head attribute defines the imperative verb to be used to identify the MoveCommand, the DirectObject attribute says which SemanticCase is the syntactic direct object of the imperative verb, and the Cases attribute says which prepositions are used to mark the other SemanticCases in English input. Symbols like <movehead> are non-terminals in a grammar used by the DYPAR parser mentioned earlier which provides pattern matching services to MULTIPAR. They expand in the course of computation (e.g. <movehead> -> move | transfer). SurfaceForm slotfillers that are not surrounded by ◇'s are SemanticCase names and tell the strategy which SemanticCase to bind the information to. Consider the DirectObject case of the Icf-Canonical SurfaceForm in the MoveCommand. The strategy that knows how to parse this kind of surface form will be passed an instance of the movecommand entity definition and an input segment to work on. When it finds a noun phrase in the input unmarked by prepositions, it will check the surface form to find out what SemanticCase it should be trying to fill. Since the Object SemanticCase can be filled by either a FileObjDesc or DirectoryObjDesc, it will try to parse the unmarked segment as each of these. This will result in calls to strategies that know how to handle the SurfaceForms in the entity definitions of FileObjDesc and DirectoryObjDesc respectively. In general, an entity may

---

[2] FileObjDesc means file object description; other abbreviations are similar

have more than one surface form corresponding to different forms of surface expression for the same underlying semantic cases.

- **InstanceTemplate**: This information is used when a strategy has finished parsing an entity. It tells MULTIPAR the final representation to use for the entity instance thus produced. It is essentially a method for reformatting, canonicalizing, and pulling information out of subordinate entity instances to compose the current one. The slotfillers in the InstanceTemplate act as directions to a routine that uses the bindings to the SemanticCases produced by the strategy. A single word means the value of the slot is exactly what is bound to the SemanticCase. A list without "or" means the value of the slot is whatever is found in the slot with that name in the InstanceTemplate bound to the SemanticCase. (e.g. IsA in Object says look at the IsA field in whatever kind of entity is bound to Object — if the field isn't found, the slot = nil). Finally, directions that have one or more "ors" act as deterministic disjuncts. Each instruction is followed until a non-nil value is found. In this way, "move [c410jf90]foo to my directory" produces the same InstanceTemplate as "move foo from [c410jf90] to my directory." In the former, the directory name for the source of the move is found in the FileObjDesc InstanceTemplate bound to the Object SemanticCase. In the latter, the directory name for the source of the move is found in the DirectoryObjDesc InstanceTemplate bound to the source SemanticCase. A final note about InstanceTemplates: each has a special slot called "deviations" which has no corresponding SemanticCase. This slot acts as a repository of information about the recovery actions taken by strategies.

Entity definitions apply to objects as well as actions. For instance, the entity definition for file objects is:

```
(EntityName     FileObjDesc
 SemanticCases  (
        Quantifier          Pattern
        FileName            FileNameDesc
        FileExtension       FileExtensionDesc
        FileDirectory       DirectoryObjDesc
        FileLogicalDevice   LogicalDeviceObjDesc
        Owner               OwnerDesc
        Size                SizeDesc)
 Constraints    ()
 SurfaceForms   (
        (SFName Ncf-Canonical
         Head           <FOD-head-forms>
         NonAVCases     (Quantifier <quant> !q)
         AttValCases    (Simple FileName or FileExtension or Owner or Size
                         Complex FileDirectory or FileLogicalDevice))
        (SFName Ncf-System
         Head           FileNameDesc
         Bind           FileName
         NonAVCases     (Quantifier <quant> !q)
         AttValCases    (Simple FileExtension or Owner or Size
                         Complex FileDirectory or FileLogicalDevice)))
 InstanceTemplate (
        IsA  'FILE
        Deviations Deviations
        Name (FileName in FileName)
        Extension (Value in FileExtension or FileExtension in FileName)
        Directory (DirectoryName in FileDirectory or
                   DirectoryName in FileName)
        LogicalDevice (Value in FileLogicalDevice or
                       LogicalDeviceName in FileDirectory or
                       LogicalDeviceName in FileName)
        Description (
                Quantifier Quantifier
                Owner Owner
                Size Size)))
```

This example illustrates some points not apparent in the entity definition for MoveCommand. In particular, note that the Quantifier SemanticCase is filled by a pattern, rather than another entity, i.e. it is primitive. Also, FileObjDesc has two SurfaceForms. Ncf-Canonical is designed to recognize input like "the files[3] with extension lsp in directory [c410jf90]", while Ncf-System is designed to recognize descriptions including proper names, such as "foo.lsp in [c410jf90]".

---

[3] <FOD-Head-forms> expands to file | files | program | programs

8.3. An annotated example

To make clear how all the components of MULTIPAR combine and interact to parse both well-formed and ungrammatical input, we include the following extended example. In the discussion that follows, function calls and strategy names are written in boldface type; entity slots and values are given in italics. The notation "$word_1$...$word_n$" refers to the portion of the input utterance starting with $word_1$ and ending with $word_n$.

Consider the behaviour of the system when the user requests a parse of:

Move the accounts directory the file Data3[4]

Step 1. Initialization.

Before any part of the input is examined, the control tree is initialized as in Figure 1, label (a)[5]. Here, MULTIPAR sets up a branch for each of the commands in its current vocabulary. Note that no flexibility increment is added to the initial value of zero because no deviation has occurred. The control chooses the first level 0 branch on the agenda for continuation.

[FIGURE 1 GOES HERE]

Step 2. Try to parse "Move...Data3" as a *DeleteCommand*.

ParseEntity (Figure 1,(b)) is a function that maps entity types to strategies. A request for a command entity could result in the trial of a number of different strategies. At present, only top-down versions of the parsing strategies exist, and calls to **ParseEntity** that look for commands are always mapped into calls to the imperative caseframe strategy (Figure 1,(c)). **ImperativeCaseFrame-Strat** will be unable to find an appropriate verb for the *DeleteCommand* entity and will fail without scheduling any alternate branches (i.e. this can be viewed as a non-recoverable error for the top-down strategy). Failure means that processing is continued by the control structure which eliminates this branch (Figure 1,(d)) and chooses another (Figure 1,(e)). Of course, the new branch also contains a call to **ParseEntity**; this call is mapped as above (Figure 1,(f)).

Step 3. Try to parse the input as a *MoveCommand*.

ImperativeCaseFrame-Strat knows how to use the *lcf-canonical SurfaceForm* to interpret the input. In the *MoveCommand* entity definition shown in section 8.2, this is the only *SurfaceForm*. As

---

[4]The grammatically correct version of this sentence is "Move to the accounts directory the file Data3."

[5]Hereafter, simply Figure 1,(a).

linguistic coverage is extended to include, for example, declaratives and interrogatives, new *SurfaceForms* must be defined. **ImperativeCaseFrame-Strat** could be expanded to interpret all top-level forms or each form could be provided with its own "expert".

The first action that **ImperativeCaseFrame-Strat** takes is to use the *Head* field in the *SurfaceForm* to search for a legal verb.[6] It finds "Move" and the unparsed segment is reduced to "the accounts directory the file Data3". The next step is to call a routine to fill the *SemanticCases* of the *MoveCommand*.

Step 4. Use **ImperativeCaseFrame-Cases** to fill *MoveCommand*'s *SemanticCases*.

ImperativeCaseFrame-Cases (Figure 1,(f)) is not a strategy itself but only a part of the top-down imperative caseframe strategy. The distinction is important because the responsibility of a strategy is to return an "instance list," i.e. one or more instantiated *InstanceTemplates*. **ImperativeCaseFrame-Cases** will return lists of consistent *SemanticCase* bindings which **ImperativeCaseFrame-Strat** will use to fill in *InstanceTemplates* when building its instance list.

When filling cases we impose no order on their appearance in the utterance, nor do we fill required cases first (doing so would eliminate possible parses at flexibility levels greater than 0). As we try to expand a partial parse the still-unfilled cases may be constrained in the kinds of values they can take on by the values bound to those cases already filled. The *Constraints* field of the entity definition specifies the requirements. Of course, at this point no cases have been filled and no constraints apply. The unparsed segment, "the accounts directory the file Data3", is examined in two ways:

a. The first case we try to fill is the direct object which is unmarked. (Figure 1,(g) and (h))

b. The first case we try to fill is one of the marked cases. (Figure 1,(i))

Consider step 4.a. We are attempting to interpret some portion of the segment as the *Object SemanticCase* of the *MoveCommand*. Since the entity definition shows that an *Object* can be an instance of either a *FileObjDesc* or a *DirectoryObjDesc*, we will try each of these in turn (Figures 2,(a) and 3,(a)).

---

[6] Keep in mind that at each step it is possible to have more than one interpretation of the input. Thus, if our input had been, "Move the file called transfer to dskb," the initial scan for a verb would have resulted in two partial parses -- one catching "move" and the other catching "transfer".

Step 5. Continue step 4.a.; try to parse *Object* as a *FileObjDesc*.

ImperativeCaseFrame-Cases wants to find a *FileObjDesc* in "the accounts directory the file Data3". To do so, it must call **ParseEntity** with a request for a noun phrase that can be interpreted as a *FileObjDesc*. Referring to the entity definition for *FileObjDesc* (section 8.2) we see that there are two *SurfaceForms* for parsing nominal caseframes. Each has its own associated strategy so this call to **ParseEntity** results in a SPLIT (Figure 2,(b)) We will examine only the path labelled **NominalCaseFrame-Strat-Canonical** (Figure 2,(c)).

[FIGURE 2 GOES HERE]

Step 6. Find the *Head* and *Quantifier* of the noun phrase.

As with imperative caseframes, the first action taken to fill a nominal caseframe is to locate the *Head*. **NominalCaseFrame-Strat-Canonical** finds "file" as a possible head and breaks the remainder of the input into prenominal and postnominal segments. The strategy then looks for the quantifier/determiner at the left end of the prenominal segment. The value of the *NonAVCases*[7] field is interpreted: call the pattern-matcher with the prenominal segment and the pattern "<quant>," and if the pattern-matcher returns the variable "!q" bound to some value, that value is the filler for the *Quantifier SemanticCase*. With the *NonAVCases* finished only the *AVCases* remain. "Accounts directory the" now constitutes the prenominal segment and "Data3," the postnominal segment. We call **NominalCaseFrame-Cases**, a sub-routine of **NominalCaseFrame-Strat-Canonical**, to begin filling cases from the prenominal segment (Figure 2,(c)).

Step 7. Parse the prenominal segment, "accounts directory the".

The only *FileObjDesc* case we will be able to fill from this segment of the input is *FileDirectory*. The entity definition shows that this case is an instance of a *DirectoryObjDesc*. After a sequence of calls (Figure 2,(d)) a sub-invocation of **NominalCaseFrame-Strat-Canonical** will return an instance list with a single instance:

```
(IsA DIRECTORY
 Name (accounts))[8]
```

Thus, we return to step 6 (Figure 2,(c)) with one case filled and the word "the" unused.

---

[7] Non-Attribute-Value Cases

[8] The *InstanceTemplate* has many more fields in it; only those with a non-nil value are shown.

Step 8. Parse the postnominal segment, "Data3".

We are interested in extending any partial parses created in step 7 by using the postnominal segment to fill any unbound *SemanticCases*. Figure 2,(e) shows that a succession of calls results in filling the *FileName* using the Attribute-Value strategy.

Step 9. Return to step 4, Figure 1,(g).

Steps 7 and 8 produced one consistent set of bindings for two *SemanticCases* in the *MoveCommand* with one word leftover. Although there are unfilled cases, we have run out of input on the right. Thus, the instance list of *FileObjDescs* has only one element:

```
(IsA    FILE
 Name   (Data3)
 Directory  (accounts)
 Description (
       Quantifier (the)))
```

Referring to step 4.a. (Figure 1,(g)), the possible interpretations of the input such that the *Object* is a *FileObjDesc* have been exhausted. It remains to examine what happens when we look for an *Object* that is a *DirectoryObjDesc*. Again, we will call **ParseEntity**, split and suspend one of the calls, and examine the branch labelled **NominalCaseFrame-Strat-Canonical** (Figure 3,(a) through (c)).

[FIGURE 3 GOES HERE]

Step 10. Continue step 4.a.; try to parse the *Object* as a *DirectoryObjDesc*.

We pick up the *head*, "directory," and the quantifier/determiner case as in step 6. This leaves the word "accounts" in the prenominal segment and "the file Data3" in the postnominal segment. The Attribute-Value strategy finds "accounts" as the *Name*. No other *DirectoryObjDesc* cases can be filled, so this step returns:

```
(IsA DIRECTORY
 Name (accounts)
 Description (
       Quantifier (the)))
```

Step 11. Return to step 4.

Consider Figure 4,(a) which corresponds to the "OR" in Figure 1,(f). The computations shown in Figures 2 and 3 have given us two partial parses with the direct object filled; once by a *FileObjDesc* with the word "the" leftover and no input left to fill the other cases of the *MoveCommand* (Figure

4,(b)), and once by a *DirectoryObjDesc* with "the file Data3" leftover (Figure 4,(c)). As we try to extend this second partial parse, we have only marked cases remaining in the *SurfaceForm*. Since there is no marker at the beginning of the remaining segment we have encountered our first violated expectation. The recovery action associated with this failure is to hypothesize the existence of the missing case marker. Thus, for each of the three remaining *SemanticCases* we schedule a continuation that charges three flexibility points for the deviation (Figure 4,(d)). Note that if some other branch of the search tree with cumulative flexibility less than three succeeds in consuming the entire input segment, the branches just spawned will never be reactivated.

[FIGURE 4 GOES HERE]

Step 12. Return to step 4.b. (Figure 4,(e))

Consider what happens when we try filling the marked cases of the *MoveCommand* first (Figure 1,(i) corresponding to Figure 4,(e)); the situation is identical to the one just outlined. For all cases other than the direct object, the segment "the accounts directory the file Data3" must have a left marker. Since it has none, we hypothesize a branch for each marked case at the current flexibility level plus three (Figure 4,(e)).

Step 13. All the possibilities for the *MoveCommand* have been examined.

We have succeeded in finding two ways to fill the cases of the *MoveCommand*. Before we can return the partial parses from **ImperativeCaseFrame-Cases** to **ImperativeCaseFrame-Strat** we must check whether the *Required* cases, as specified by the *Constraints* field, have been filled. Indeed, each of the partial parses is missing the required *Destination* case, a violated expectation. The recovery action associated with this error is to suspend each of these partial parses and charge two flexibility points per missing required case for their continuation (Figure 4,(f) and (g)). Since no parses had all the required cases, the level 0 continuation of **ImperativeCaseFrame-Cases** returns a failure signal to **ImperativeCaseFrame-Strat**.

ImperativeCaseFrame-Strat returns an instance list with a single instance whose *Source* and *Destination* fields are nil. This signifies that the only part of the strategy that succeeded at level 0 was finding the verb. Since there is unused input, the top-level of MULTIPAR interprets this instance as a failure and signals this to the control.

Step 14. Exhaust level 0 of the agenda looking for a non-deviating parse.

The control structure takes over and continues in turn each branch suspended at level 0. Those containing requests for imperatives (Figure 1,(j)) fail immediately as in step 2. The other level 0 branches were left suspended by the SPLITs in Figures 2 and 3; these also fail.

Step 15. Exhaust levels 1 and 2 of the agenda.

Having tried all the branches in level 0 without success, the flexibility level is incremented and the control structure tries to choose a path suspended at level 1. Our example did not spawn any level 1 branches (single spelling corrections), so the flexibility level is incremented again. There are two branches at level 2, both in the same predicament (Figure 4, (f) and (g)); each has a missing required case and leftover input. If there had been no leftover input (as in "Move foo"), they would have succeeded at this level.[9] However, since no further recovery actions apply, each of these branches fails without adding to the control tree.

Step 16. The control increments the flexibility level to 3.

There are two sets of branches at this level:

a. The *Object* case is filled and the missing marker has been hypothesized before "the file Data3". (Figure 4,(d))

b. No cases are filled and the missing marker has been hypothesized before "the accounts directory the file Data3". (Figure 4,(e))

Step 17. Continue 16.a. (Figure 4,(d))

The *Object* case of the *MoveCommand* has been bound to a *DirectoryObjDesc* with the *Name* field bound to "(accounts)". Hypothesizing the appropriate kind of marker for each of the remaining cases gives:

a. *Source*: Move the accounts directory [from] the file Data3.

b. *Destination*: Move the accounts directory [to] the file Data3.

c. *Location*: Move the accounts directory [in] the file Data3.

The single recovery action of hypothesizing a missing marker is not enough for any of these branches to succeed. Each would be rescheduled at least one more time. If MULTIPAR allowed the

---

[9] Although consuming the entire input would have guaranteed success, note that if some branch with cumulative flexibility of 0 or 1 had succeeded, these branches would never have been retried.

control structure to search the space indefinitely, 17.a. would eventually succeed at flexibility level 8 (3 points for a missing marker, 3 points for a constraint violation[10], and 2 points for a missing required case). 17.b. would eventually succeed at level 6 (1 missing marker, 1 constraint violation) and 17.c. at level 8 (1 missing marker, 1 missing required case and 1 constraint violation).

Step 18. Continue 16.b. (Figure 4,(e))

There are three branches remaining, one for each of the marked cases in the *MoveCommand*, with no cases yet filled. Allowing indefinite expansion, the following would occur:

Source:
- a. Move [from] the accounts directory the file Data3.
  eventually succeeds at + 5; 3 for missing marker, 2 for missing case
- b. Move [from] the accounts directory [to] the file Data3.
  eventually succeeds at + 8; 2 missing markers, 1 required case
- c. Move [from] the accounts directory [in] the file Data3.
  eventually succeeds at + 13; 2 missing case markers, 1 constraint violation and 2 missing required cases

Destination:
- a. Move [to] the accounts directory the file Data3.
  *** SUCCEEDS at this level (level 3) ***
- b. Move [to] the accounts directory [from] the file Data3.
  eventually succeeds at + 8; 2 missing case markers and 1 missing required case
- c. Move [to] the accounts directory [on] the file Data3.
  eventually succeeds at + 13; same as *Source* c.

Location:
- a. Move [in] the accounts directory the file Data3.
  eventually succeeds at + 5, 1 missing marker and 1 missing case
- b. Move [in] the accounts directory [to] the file Data3.
  eventually succeeds at + 8, 2 markers, 1 case
- c. Move [in] the accounts directory [in] the file Data3.
  eventually succeeds at + 13, same as *Source* c

Step 19. A successful path is found at level 3.

Hypothesizing the existence of a marker for the *Destination* enables **ImperativeCaseFrame-Strat** to continue the second branch of Figure 4,(e). Now "the accounts directory" can be picked up as the *Destination* and "the file Data3" as the unmarked direct object. Since both required cases are bound

---

[10] In *Constraints:(Object DirectoryObjDesc > Source LogicalDeviceObjDesc)*.

and no input remains, MULTIPAR return the following instance as its representation of the input:

```
(Action MOVE
 Deviations (MissingMarker Destination)
 Source (
       IsA FILE
       Name (Data3)
       Description (
              Quantifier (the)))
 Destination (
       IsA FILE
       Name (Data3)
       Directory (accounts)
       Description (
              Quantifier (the))))
```

9. Applications to Speech Input

9.1. Special characteristics of speech input

Spoken natural language input to a computer is characterized by large amounts of error, much more error than found in typed natural language input. These errors stem both from the speaker and from imperfect recognition. Given the large potential advantages to achieving robust speech recognition, we attempted to further extend and develop our flexible parsing techniques by applying them to spoken input. This proved to be a fruitful area of application, as described below. The work ran from early 1985 to the end of the contract in June 1986. It was done in conjunction with a concurrent DARPA-funded speech project in the Carnegie-Mellon Computer Science Department. Further details can be found in[8].

Not surprisingly, it is not possible to apply techniques developed for parsing typed natural language to spoken input in a completely straightforward manner. We list some of the problems below. We assume the existence of a *speech recognizer* that will transform a spoken input into a *word lattice* — a set of hypothesized words that may be present in the input, together with their starting and ending positions in the input signal and the probability of each word being correct. In general, there will be several competing word hypotheses for each point in the input signal.

- **lexical ambiguity**: More than one word may be produced by the speech recognizer for a given segment of speech. If the ambiguities were simply between different word choices, this problem could be handled by the techniques used to handle word sense ambiguity in natural language processing (e.g. "bank" may mean a place to put money, the side of a river, an action of placing trust, an action of tilting a vehicle left or right, etc.). However, not only can multiple words be hypothesized, but the competing hypotheses can occur at overlapping, adjoining, or separate segments of the input signal, producing no consistent set of word boundaries. There is no parallel phenomenon in processing typed natural language.

- **probability measures**: Speech processing systems typically provide information about the relative likelihood of the correctness of each of their word hypotheses. These probabilities or scores are based on criteria such as the quality of the match between speech signal and phonemic dictionary expectations. Since a speech recognition system may hypothesize many words for the same segment of speech, and since the scores for these words may differ considerably, these scores are important in limiting the search of a parser. However, there is no natural way to make use of such likelihood scores through most of the current natural language processing techniques.

- **unrecognized words**: Because of hurried pronunciation or co-articulation effects, a speech recognizer may completely fail to recognize some of the words in an input utterance. The missed words are usually (though not always) short, unstressed, "function" words rather than longer "content" words. This kind of omission is not handled by standard natural language processing techniques.

- **ungrammatical input**: In addition to the word omissions that are artifacts of imperfect word hypothesization by a speech processor, spoken input tends to contain more real grammatical deficiencies than typed input. Once spoken, words cannot be easily retracted, but typed utterances can be corrected if the user notices the error in time. Thus, fail-soft techniques for recovery from all kinds of grammatical errors in natural language processing are particularly pertinent when extended to the interpretation of spoken input.

These difficulties argue against the simplistic approach of taking a speech-recognition module and attaching it to a traditional natural language analyzer designed to work from words entered as unambiguous ASCII characters. No matter how good each may be in isolation, the two will not integrate successfully if the latter cannot provide semantic expectations to the former, cannot handle massive lexical ambiguity, or cannot tolerate errors of recognition and of grammatical deviation. Moreover, with adequate integration, feedback from a natural language analysis component can substantially improve the performance of a connected speech recognizer. This performance enhancement is badly needed since no present connected speech recognition method comes close to human abilities. And even humans fail to recognize a substantial fraction of function words extracted from their surrounding context. The application of linguistic knowledge and semantic expectations through natural language analysis techniques is thus needed to complement acoustic recognition methods by constraining the set of possible (and sensible) interpretations of the words in an input utterance.

9.2. Applying caseframes to speech input

In this subsection, we discuss the ways in which we have adapted our caseframe-based techniques developed in the context of typed input. The two key observations in performing these adaptations were:

- caseframes of the kind we have described contain the right amount of information at the right level of abstraction to parse restricted-domain spoken input;

- the algorithms that have been developed for using such caseframes in parsing typed natural language input are unsuitable for spoken input because the algorithms rely on the presence of small function words that are recognized at best unreliably by word hypothesizers. In particular, many of the strategies that we have used for typed input (including in FlexP, CASPAR, and MULTIPAR) rely on the presence of prepositions, which tend to be short function words.

Based on these observations, we have produced a trial implementation of an approach to parsing spoken input from semantic caseframes. The approach applies caseframes to the input, but does it in a novel way. The essence of the approach is to:

1. examine the lattice of words hypothesized by the speech recognizer for those that correspond to caseframe headers

2. combine all the caseframes corresponding to the words found in all semantically and syntactically plausible ways

3. for each caseframe combination thus formed, attempt to account for the gaps between the caseframe header words that were involved in its formation by parsing words from the gaps against empty semantic and syntactic roles in the caseframe combination

4. select as the final parse of the input those caseframe instances that best account for the input, based on how much of the input they cover and the acoustic scores of the words in that parse.

This multi-stage approach avoids the problems of the caseframe parsing algorithms for typed input by anchoring the parse on caseframe headers. Caseframe headers are verbs (for clausal caseframes) and nouns (for nominal caseframes). These are content bearing words that tend to be stressed in speech and are often multi-syllabic. Both these qualities improve their chances of recognition above that of short, unstressed function words. The points around which the parse is anchored are thus likely to be highly correlated with the words in the input that are most certain acoustically.

An advantage associated with working from caseframe headers is that the resulting caseframe combinations form a ready-made semantic interpretation of the input. The interpretation is typically incomplete until it is filled out in the subsequent gap-filling stage. However, if the recognition of some or all of the remaining words is so poor that the semantic interpretation is never fully completed, then the parser still has something to report. Depending on the application domain, a skeleton interpretation of this kind could be sufficient for the application, or would at least form the basis of a focussed request for confirmation or clarification to the user[6].

In what follows, we examine in more detail our implementation of the approach outlined above. The flexible parser we have implemented for this application operates in the context of a complete speech understanding system that handles continuous speech with a 200 word vocabulary in an electronic mail domain.

9.2.1. Example caseframes

Because of experience gained in previous phases of the project, the caseframes used for speech parsing are somewhat different from those used previously. In particular, they incorporate lessons learnt in making the caseframe definitions more intuitive in terms of the restricted domain which they describe. Accordingly, we describe the formalism used.

```
#S(ED
  Name ForwardAction
  Type verb
  SC (
    #S(SC
      Name    Agent                    ; the sender
      InstanceOf (MailAdrDesc)
      SyntaxCase (subject)
      )
    #S(SC
      Name    MsgObj                   ; a message
      InstanceOf (MsgObjDesc)
      SyntaxCase (DO)
      )
    #S(SC
      Name    MsgRecipientObj          ; the receiver
      InstanceOf (MailAdrDesc)
      SyntaxCase (IO Prep0)
      CaseMarker (to)
      )
    #S(SC
      Name    CCRecipientObj           ; the CarbonCopy
      InstanceOf (MailAdrDesc)         ; receiver
      SyntaxCase (Prep0)
      CaseMarker (ccing copying)
      )
    )
  Constraints #S(constraint
    requiredSC (MsgObj MsgRecipientObj Agent)
    )
  HeadForms (forward resend)
  )
```

Figure 1: Caseframe for forward

Figure 1 defines the *forward* action of an electronic mail system. The caseframe is identified as a verb or clausal caseframe corresponding to the verbs (HeadForms) "forward" or "resend". It also has four cases: Agent (the person doing the sending), MsgObj (the message being forwarded),

MsgRecipientObj (the person the message is being forwarded to), and CCRecipientObj (the people who are to get a copy of the forwarded message). The MsgObj case must be filled (InstanceOf) by a MsgObjDesc (defined by another caseframe, see below), and the other cases must be filled by a MailAdrDesc (the caseframe representing a person or "mail address"). All the cases are required, except CCRecipientObj, which is optional. In addition, to this purely semantic information, the caseframe contains some syntactic information: the Agent case is manifested as the syntactic subject; MsgObj as the direct object (DO); MsgRecipientObj as either the indirect object (IO) or as the object (PrepO) of a prepositional phrase, whose preposition (CaseMarker) is "to"; CCRecipientObj as a prepositional phrase with "prepositions" either ccing or copying.

```
#S(ED
  Name MsgObjDesc
  Type Noun
  SC (
    #S(SC
      Name    Descriptors
      Pattern (new recent old unexamined examined)
      SyntaxCase (prenominal)
      )
    #S(SC
      Name    Determiners
      Pattern (the this that any a every)
      SyntaxCase (prenominal)
      )
    #S(SC
      Name    MsgOriginObj       ; where the mail
      InstanceOf (MailAdrDesc) ; came from
      CaseMarker (from)
      SyntaxCase (PrepO)
      )
    #S(SC
      Name    TimeObj
      InstanceOf (HourDesc MonthDesc DayDesc)
      CaseMarker (from before after since on at)
      SyntaxCase (PrepO)
      )
    )
  HeadForms (message mail)
  )
```

Figure 2: Caseframe for message

In addition to actions, we also use caseframes to describe objects Figure 2 shows a nominal caseframe for the message object of our electronic mail system. This has essentially the same form as the verb caseframe, except that its HeadForms correspond to the head nouns of a noun phrase describing an electronic mail message. In addition, the Descriptors case has a new SyntaxCase, prenominal, which implies that the elements of Pattern (new, recent, etc.) may appear in the adjective position in this caseframe.

9.2.2. The word lattice

The input to our caseframe speech parser can be viewed as a two-dimensional lattice of words. Each word has a begin time, an end time, and a likelihood score. The begin/end times state where the word was detected in the utterance. The score indicates how certain we are that the word is correct, based on acoustic/phonetic information. In the sample lattice below, the horizontal dimension is time, and the vertical dimension corresponds to certainty of recognition of individual words by the speech recognizer generating the lattice. This word lattice was constructed by hand for demonstration purposes.

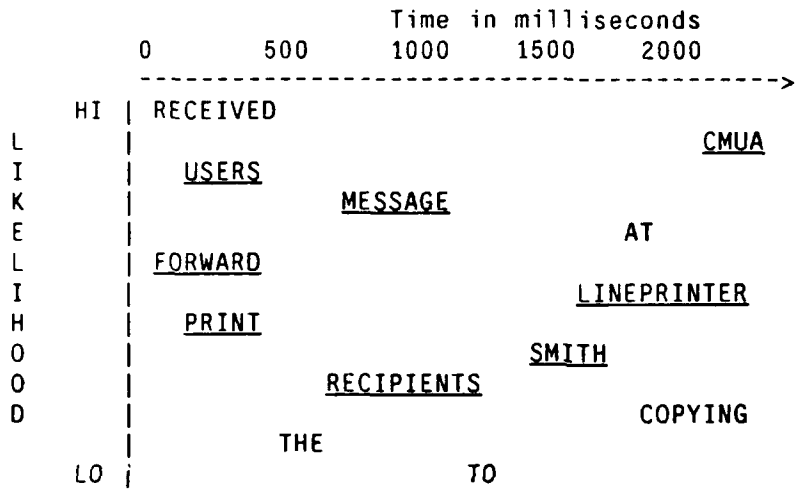


Figure 3: A simplified word lattice containing different kinds of words. Header words are underlined

9.2.3. Header combination

To start its processing of an input word lattice, the parser selects from the word lattice all header words above a recognition likelihood threshold. These headers correspond to caseframes, but only some combinations of the hypothesized caseframes are possible in the domain. To calculate the legal caseframe combinations, a set of phrase structure rules were derived that apply at the frame level (rather than at the more detailed word level).

To make matters more concrete, let us refer to the sample lattice above. In this lattice, the underlined header words would be combined to form the nuclei of sentences like: *"Forward message Smith CMUA"* and *"Print message lineprinter."* Caseframes can combine in this way if one is of the right type (as defined by the InstanceOf attribute for the case) to fill a case of another. When combining caseframes associated with header words, the parser also uses knowledge about word order to limit the possible combinations. In our example, the *forward* caseframe (as defined in Figure 1) has a slot for a MsgObjDesc as a DO. The order restrictions built into the parser only allow for the direct object (DO) after the verb. The *message* caseframe (Figure 2) fulfills these requirements. It is a

MsgObjDesc, whose HeadForm "message" occurs after the *forward* caseframe HeadForm "forward" in the lattice. Thus the two can be combined, as long as the constraint of the required MsgRecipientObj can be satisfied (by "Smith").

Each time a valid sequence of headers is found, it is given an overall likelihood score and merged with the previous ones. At the end of the header combination phase, we have a list of ordered partial phrases, containing all the legal sequences of header words that can be found in the word lattice. Each partial phrase is represented as a set of nested caseframe instances. For instance, three combinations would be formed from the header words:

Forward message Smith CMUA

and these would have the nesting structure:

```
[ForwardAction
  HeadForm FORWARD
  MsgObj [MsgObjDesc
            HeadForm MESSAGE]
  MsgRecipientObj [MailAdrDesc
                     HeadForm SMITH
                     Host  [LocationDesc
                              HeadForm CMUA]]]

[ForwardAction
  HeadForm FORWARD
  MsgObj [MsgObjDesc
            HeadForm MESSAGE]
  CCRecipientObj [MailAdrDesc
                     HeadForm SMITH
                     Host  [LocationDesc
                              HeadForm CMUA]]]

[ForwardAction
  HeadForm FORWARD
  MsgObj [MsgObjDesc
            HeadForm MESSAGE
            MsgOriginObj
               [MailAdrDesc
                  HeadForm SMITH
                  Host  [LocationDesc
                           HeadForm CMUA]]]]
```

where square brackets indicate caseframe instances and the nesting is conveyed by textual inclusion.

A routine to check word junctures is used during the header combination phase. Whenever two header words are combined for a partial phrase, the juncture between these words is checked to ascertain whether they overlap (indicating an illegal combination), abut, or have a gap between them (indicating significant intervening speech events). This check also enables the parser to deal efficiently with co-articulated phonemes as in *"some messages"*. These phonemes are merged in

pronunciation, resulting in a pair of overlapping word candidates that are nevertheless valid. These word juncture checks comprise a top-down feedback mechanism to improve the speech recognition.

9.2.4. Casemarker connection

Once caseframe combinations have been formed, the next step is to fill in the gaps between the words of the corresponding partial phrase. We take each combination in turn, starting with the one with maximal-likelihood. The caseframe speech parser first tries to fill in casemarkers, which are usually prepositions.

Let us continue our example with the first of the header combinations formed from the phrase *"Forward message Smith CMUA"*. For this phrase, casemarkers may appear before the prepositionally marked cases *"Smith"* and *"CMUA"*. The requirement that the casemarkers must appear between the header words of the containing and contained caseframes is a strong constraint on the possible locations of the casemarkers. There are generally strong limitations on what words could possibly serve as markers for these cases. In our example, using information from the caseframe definitions of the previous section, the parser would thus try to verify one of the words *"to" "from"* between *"message"* and *"Smith"* and one of the words *"on"* or *"at"* between *"Smith"* and *"CMUA"*.

Whenever a set of words are predicted by the parser in a given segment of the input, a word verification module is called. This module has knowledge of the complete word lattice. A word that matches the prediction is sought from the lattice in the specified gap. In addition, the acoustic-phonetic data is consulted to give an indication whether the word is a perfect fit for the gap, a left or right anchored fit, or if there are intervening significant speech events on the left or right. This information allows the parser to determine how much of the input has been accounted for by a given partial phrase hypothesis.

Every successfully verified casemarker causes the parser to spawn another partial phrase hypothesis (analogously to the SPLIT operation in MULTIPAR). The word could be a spuriously hypothesized word, i.e. one that was "recognized" even though it was never spoken (also known as a false alarm). Therefore we leave the old partial phrase without the casemarker in the ordered list of partial phrases and merge a new partial phrase into the list. The new partial phrase is a copy of the old one, with the casemarker also filled in. A new likelihood score is computed for this phrase.

The score for a partial phrase is currently computed as the sum of the time normalized probabilities of each word divided by the time of the total utterance. Thus the probability of each word is multiplied

by the duration of the word, summed over all words and divided by the duration of the utterance. This favors longer partial phrases over shorter ones. However, even extremely low scoring long phrase candidates are favored over well scoring shorter phrases. Alternative scoring procedures for partial phrases could recognize the tradeoff between long, low scoring utterances that seem to account for all the input and short phrase hypotheses with excellent scores that leave gaps in the utterance unaccounted for. An ideal scoring function would also use semantic and syntactic wellformedness as criteria.

Sometimes, none of the case markers being verified are found. This may mean that:

- the speech recognizer failed to detect the marker. Unvoiced co-articulated monosyllabic words (such as prepositions) often go undetected;
- or, the most-likely parse at the case-header level was indeed incorrect, and a lower likelihood parse should be explored to see if it is more consistent with the acoustic data.

At present only the second choice is considered, but it should be possibile to develop an enhanced verifier to re-invoke the lower level processes (acoustic analysis or word hypothesizer modules) with strong expectations (one or two words) at a prespecified window in the input. We would hope that such a process could detect words missed in a more cursory general scan — and thus use semantic and syntactic expectations to drive the recognition of the most difficult segments of the speech form. If the verifier were to return with a recognized case marker, but a low likelihood, the overall likelihood value of the next parse could make it the preferred one.

9.2.5. Prenominal filling

The next phase in processing fills in the prenominal sections of the partial phrases. The parser looks for prenominals in the following order:

*Predeterminer Determiner Ordinal Cardinal Adjective**

A lexicon associates each potential prenominal word with the correct type. Thus we first look for all possible predeterminers (e.g. 'all') within the available gap before the corresponding header word. Again the successful verification of such a prediction spawns a new partial phrase, just as described for casemarkers. The old partial phrase remains in the list as a precaution against false alarms. It should be noted that remaining old phrases accounting for less input receive a lower global likelihood value because unaccounted for input is penalized.

Then determiners are examined. In our example, the determiner "the" will successfully be found to modify the message caseframe. The other prenominal types are filled in the same way. Post-nominal modifiers (i.e., prepositional phrases) are parsed by the caseframe instantiation method above, as nominal and sentential caseframes are treated in much the same way.

9.2.6. Extending coverage to simple questions

Although we have not made completeness of syntactic coverage a focus in our work on speech parsing, we made some simple extensions to gain some idea of the difficulty in syntactic extension. In particular, we extended the system to deal with simple interrogatives as well as imperatives and declaratives. No changes to the caseframes themselves were necessary, just to the parsing algorithm. We introduced a separate stage in processing to look exclusively for question words. These words may be the standard wh-words (who, what, when, ...) or sentence-initial auxiliary verbs to indicate a yes/no question (do, does, is, will, ...).

The word order rules in the header combination phase also required extension. These rules now have to allow fronted cases

What messages did Smith send

and questions where the HeadForm of the case is collapsed into a question word

Who sent this message

. Finally, we added a new module to fill auxiliary verbs in the correct locations. It operates just like the casemarker connection module and will not be described further here. By providing the parser with constraints governing the agreement of subject/verb, of auxiliary verb/main verb, and of prenominal/noun, the number of plausible alternatives is kept low. The relative ease of this extension is another demonstration of the advantages of the multi-strategy approach to parsing that we have developed and used.

References

1. Ball, J. E. and Hayes, P. J. Representation of Task-Independent Knowledge in a Gracefully Interacting User Interface. Proc. National Conference of the American Association for Artificial Intelligence, Stanford University, August, 1980, pp. 116-120.

2. Carbonell, J. G. and Hayes, P. J. "Recovery Strategies for Parsing Extragrammatical Language". *Computational Linguistics 10* (1984).

3. Carbonell, J. G. and Hayes, P. J. Dynamic Strategy Selection in Flexible Parsing. Proc. of 19th Annual Meeting of the Assoc. for Comput. Ling., Stanford University, June, 1981, pp. 143-147.

4. Erman, L. D., and Lesser, V. R. HEARSAY-II: Tutorial Introduction and Retrospective View. Carnegie-Mellon University Computer Science Department, 1978.

5. Fain, J. E., Carbonell, J. C., Hayes, P. J., and Minton, S. N. MULTIPAR: a Robust Entity-Oriented Parser. Proceedings of Seventh Annual Conference of the Cognitive Science Society, University of California, Irvine, August, 1985.

6. Hayes P. J. A Construction Specific Approach to Focused Interaction in Flexible Parsing. Proc. of 19th Annual Meeting of the Assoc. for Comput. Ling., Stanford University, June, 1981, pp. 149-152.

7. Hayes, P. J. Entity-Oriented Parsing. COLING84, Stanford University, July, 1984.

8. Hayes, P. J., Hauptmann, A. G., Carbonell, J. G., and Tomita, M. Parsing Spoken Language: a Semantic Caseframe Approach. COLING86, Bonn, August, 1986.

9. Hayes, P. J., and Reddy, R. Graceful Interaction in Man-Machine Communication. Proc. Sixth Int. Jt. Conf. on Artificial Intelligence, Tokyo, 1979, pp. 372-374.

10. Hayes, P. J., and Reddy, R. An Anatomy of Graceful Interaction in Man-Machine Communication. Carnegie-Mellon University Computer Science Department, 1979.

11. Hayes, P. J. and Mouradian, G. V. "Flexible Parsing". *American Journal of Computational Linguistics 7*, 4 (1981), 232-241.

12. Hayes, P. J. and Carbonell, J. G. Multi-Strategy Construction-Specific Parsing for Flexible Data Base Query and Update. Proc. Seventh Int. Jt. Conf. on Artificial Intelligence, Univ. of British Columbia, Vancouver, August, 1981, pp. 432-439.

13. Hayes, P. J. and Szekely, P. A. Graceful interaction through the COUSIN command interface. Carnegie-Mellon University Computer Science Department, 1983. To appear in *International Journal of Man-Machine Studies*..

14. Minton, S. N., Hayes, P. J., and Fain, J. E. Controliing Search in Flexible Parsing. Proc. Ninth Int. Jt. Conf. on Artificial Intelligence, Los Angeles, August, 1985.

Cummulative Publication List

1. Ball, J. E. and Hayes, P. J., "Representation of Task-Independent Knowledge in a Gracefully Interacting User Interface", *Proc. National Conference of the American Association for Artificial Intelligence*, Stanford University, August 1980, pp. 116-120.

2. Hayes, P. J. and Mouradian, G. V., "Flexible Parsing", *American Journal of Computational Linguistics*, Vol. 7, No. 41981, pp. 232-241.

3. Hayes, P. J. and Carbonell, J. G., "Multi-Strategy Parsing and its Role in Robust Man-Machine Communication", Tech. report, Carnegie-Mellon University, Computer Science Department, May 1981.

4. Carbonell, J. G. and Hayes, P. J., "Dynamic Strategy Selection in Flexible Parsing", *Proc. of 19th Annual Meeting of the Assoc. for Comput. Ling.*, Stanford University, June 1981, pp. 143-147.

5. Hayes P. J., "A Construction Specific Approach to Focused Interaction in Flexible Parsing", *Proc. of 19th Annual Meeting of the Assoc. for Comput. Ling.*, Stanford University, June 1981, pp. 149-152.

6. Hayes, P. J. and Carbonell, J. G., "Multi-Strategy Construction-Specific Parsing for Flexible Data Base Query and Update", *Proc. Seventh Int. Jt. Conf. on Artificial Intelligence*, Univ. of British Columbia, Vancouver, August 1981, pp. 432-439.

7. Hayes, P. J. and Szekely, P. A., "Graceful interaction through the COUSIN command interface", *International Journal of Man-Machine Studies*, Vol. 19, No. 3, September 1983, pp. 285-305.

8. Carbonell, J. G. and Hayes, P. J., "Recovery Strategies for Parsing Extragrammatical Language", *Computational Linguistics*, Vol. 101984.

9. Carbonell, J. C. and Hayes, P. J., "Coping With Extragrammaticality", *COLING84*, Stanford University, July 1984.

10. Hayes, P. J., "Entity-Oriented Parsing", *COLING84*, Stanford University, July 1984.

11. Fain, J. E., Carbonell, J. C., Hayes, P. J., and Minton, S. N., "MULTIPAR: a Robust Entity-Oriented Parser", *Proceedings of Seventh Annual Conference of the Cognitive Science Society*, University of California, Irvine, August 1985.

12. Minton, S. N., Hayes, P. J., and Fain, J. E., "Controlling Search in Flexible Parsing", *Proc. Ninth Int. Jt. Conf. on Artificial Intelligence*, Los Angeles, August 1985.

13. Hayes, P. J., Hauptmann, A. G., Carbonell, J. G., and Tomita, M., "Parsing Spoken Language: a Semantic Caseframe Approach", *COLING86*, Bonn, August 1986.

Professional Personnel

This list names professional personnel that have participated in the research effort at any time during its course:

1. Philip J. Hayes
   D Sc, 1977, "Some Association-Based Techniques for Lexical Disambiguation by

Machine".

2. George V. Mouradian
   M Ph, 1978.

3. Jill E. Fain
   M Sc, 1985.

4. Alex G. Hauptmann
   M Sc, 1986.

10. Interactions

Ongoing consultation with Dr. Jaime Carbonell, also a faculty member in the Computer Science Department of Carnegie-Mellon University, but not funded under this contract.

END

12-87

DTIC